Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

1st Workshop on Model-driven Software Adaptation

M-ADAPT'07 at ECOOP 2007

(Proceedings)

Edited by

Gordon Blair, Nelly Bencomo
Lancaster University
Robert France
Colorado State University

Michael Cebulla
TU Berlin

Bericht-Nr. 2007 – 10

ISSN 1436-9915

1st Workshop on Model-driven Software Adaptation - M-ADAPT'07
30 July, 2007, Berlin, Germany

Program Committee

Franck Barbier
Univ. of Pau, Netfective Technology, France
Benoit Baudry
IRISA, France
Fabio M. Costa
Federal University of Goias, Brazil
Eli Gjørven
Simula Research Laboratory, Norway
Gang Huang
Peking University, China
Rui Silva Moreira
UFP, INESC Porto, Portugal
Klaus Pohl
ICB, Germany
Marten van Sinderen
Univ. of Twente, The Netherlands
Arnor Solberg
SINTEF, Norway
Mario Trapp
Fraunhofer IESE, Germany
Steffen Zschaler
T.U. Dresden, Germany

Organizing Committee

Nelly Bencomo
Gordon Blair
Lancaster University, UK

Robert France
Colorado State University, USA

Welcome to the 1st ECOOP Workshop on Model-driven Software Adaptation - M-ADAPT'07

Preface

This document contains the proceedings of the 1st Workshop on Model-driven Software Adaptation (M-ADAPT'07). This workshop will place in Berlin, Germany on the 30th July 2007 and will be co-located with the 21st European Conference on Object-Oriented Programming (ECOOP'07). From a total of 15 papers submitted 10 papers were accepted classified as long and short papers depending of the relevance of their contributions. This volume gathers together all the papers accepted at M-ADAPT'07.

Keywords: Adaptation, Model Driven Engineering (MDE), Dynamic Variability, Variability Management, Runtime Models.

Motivation and Goals

Adaptability is emerging as a necessary underlying capability for many applications, particularly for areas such as environmental monitoring, disaster management and other applications deployed in dynamically changing environments. Such applications inevitably have to reconfigure themselves according to fluctuations in their environment. The unpredictability of changes in the environments and their requirements pose new challenges to Software Engineering. Current software development approaches specify the functionality of the system at design-time. Such approaches are not adequate to develop systems that dynamically adapt to environment fluctuations. As a result, innovative alternatives are required that take into account the specification of behaviour and functionality during execution. However, dynamic adaptation can lead to emergent and often inappropriate or unpredictable behaviour. The goal of this workshop is to explore how to develop appropriate model-driven approaches to model, analyze, and validate the volatile properties of the behaviour of adaptive systems in potentially volatile environments.

The workshop places strong emphasis on the cross-pollination of ideas from different researchers from different research fields including model-driven engineering, product lines and system families, computational reflection, and autonomous and self-adaptive systems. The workshop aims to establish a sound foundation for the use of model-driven techniques for software adaptation. Relevant topics and open research questions include:

– Formal notations for modeling, analyzing, and validating adaptive systems;

– Managing and modelling the dynamic variability intrinsic in the structure and behaviour of adaptive systems and their environments;

– The relevance and suitability of different model-driven approaches to monitoring and managing systems during runtime;

– Compatibility (or tension) between different model-driven approaches;

– Experience related to the use of run-time models to adapt software systems;

– Model-driven design for adaptability;

Workshop Format

Accepted papers were classified as long and short papers. The 6 long papers stated clear contributions on the topics and the 4 short papers clearly demonstrate scenarios where model-driven techniques may give benefits. The papers accepted are:

Long papers

– *Development of S&R embedded systems using dynamic adaptation* by Rasmus Adler, Daniel Schneider, and Mario Trapp.

– *Applying Architectural Constraints in theModeling of Self-adaptive Component based Applications* by Mohammad Ullah Khan, Roland Reichle, and Kurt Geihs.

– *A Model-driven Approach to the Development of Autonomous Control Applications* by Helge Parzyjegla, Michael Jaeger, and Gero Muehl.

– On *Run-time Behavioural Adaptation in Context-Aware Systems* by Javier Cámara, Gwen Salan, and Carlos Canal.

– *Modeling Software Adaptation Patterns* by Hassan Gomaa

– *Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments* by Jeremy Dubus and Philippe Merle

Short papers

– *Experiments with a Runtime Component Model* by J Ueyama, Geoff Coulson, Edmundo Madeira, Thas Batista, and Paul Grace

– *Endowing Software Components with Autonomic Capabilities Based on Modeling Language Executability* by Cyril Ballagny, Nabil Hameurlain, and Franck Barbier

– *Modelling Adaptation Policies for Self-Adaptive Component Architectures* by Franck Chauvel and Olivier Barais

– *A Reconfiguration Mechanism for Statechart Based Components* by Xabier Elkorobarrutia, Gaiuria Sagardui, and Xabier Aretxandieta

A primary deliverable of the workshop is a report that outlines (1) the research issues and challenges in terms of specific research problems in the area, and (2) a synopsis of existing model-based solutions that target some well-defined aspect of monitoring and managing the execution of systems.

Related Events

A similar workshop, Models@runtime'06, was held at MODELS 2006 in Italy. The topic of this event was the use of model-driven techniques to provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns. It was attended by at least twenty persons. During that workshop key research questions were identified and discussed. The ECOOP workshop will use

the research questions identified during the MODELS workshop as a basis for soliciting papers and as a starting point for further discussions. Bringing the workshop to an ECOOP audience will help broaden the discussions to cover issues related to the integration of modelling techniques with other techniques typically covered at ECOOP (e.g., component-based and reflection techniques). Models@runtime 2007 will be held at MoDELS 2007 in Nasville, USA.

We would like to thank a number of people who have contributed to this event, especially the members of the program committee who acted as anonymous reviewers and provided valuable feedback to the authors: *Franck Barbier, Benoit Baudry, Fabio M. Costa, Eli Gjørven, Gang Huang, Rui Silva Moreira, Klaus Pohl, Marten van Sinderen, Arnor Solberg, Mario Trapp*, and *Steffen Zschaler*. We also thank to Workshop Chairs *Peter Pepper Arnd* and *Poetzsch-Heff* and specially *Michael Cebulla* for the organization and patience dealing with the organization of the workshops. Last but not least, the authors of all submitted papers are thanked for helping us making this workshop possible.

June, 2007

Gordon Blair, Nelly Bencomo
Lancaster University, UK
Robert France
Colorado State University, USA

CONTENTS

Long Papers

Long Papers

Development of Save and Reliable Embedded Systems using Dynamic Adaptation

Rasmus Adler, Daniel Schneider, Mario Trapp
Fraunhofer Institute Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, 67663 Kaiserslautern
Germany
{rasmus.adler | daniel.schneider | mario.trapp}@iese.fraunhofer.de

ABSTRACT
A major application of dynamic adaptation is the development of safe and reliable embedded systems. In contrast to classical redundancy approaches dynamic adaptation can react much more flexible to different kinds of errors including changes in the environment. Moreover dynamic adaptation can usually be realized much more cost-efficient than classical redundancy or fault-tolerance mechanisms. Using dynamic adaptation for developing dependable systems requires means to explicitly specify the adaptation behavior and to analyze the effects of dynamic adaptation on system reliability and particularly safety. However, these activities are very complex and error prone and hence pose the need for a sound and seamless engineering support. For this reason, this position paper points out some of the lessons we have learned over the last years of applying and advancing dynamic adaptation for the development of safe and reliable adaptive systems. We furthermore discuss and classify current achievements in research and practice and derive corresponding future research challenges.

1. Introduction
Software systems are becoming more and more an immanent part of human life. Especially the amount of embedded software systems is constantly increasing in a huge, and still growing, number of different domains. In many cases, such systems are deployed to handle complex tasks in fields where malfunctions are to be considered critical. Such areas comprise, for instance, industrial process control, flight control systems and automotive systems. Therefore, it is important that these systems are developed to provide a high quality of service, particularly with regard to safety, which can be described as the absence of catastrophic consequences on the user(s) and the environment, and high reliability, which can be described as the continuity of correct service.
There exist different means, like fault prevention, fault tolerance, fault removal and fault forecasting, to attain higher safety and reliability in current software engineering approaches. These means certainly help to amend those characteristics in software systems, but are not sufficient in domains where high dynamics come into play and/or failures must be compensated at runtime. Two possible approaches exist to tackle this problem: a) provide sufficient redundancy by means of additional (physical) devices or b) make systems adaptive so that they are capable to compensate failures by runtime adaptation. Whereas the first approach provides the higher degree of dependability, the second approach is less expensive and brings also the advantage of making systems more flexible with respect to other QoS attributes (i.e. to react to varying resources or changing requirements). Moreover dynamic adaptation is the more flexible approach so that it is also easily possible to realize conventional safety and reliability patterns based on dynamic adaptation.
We have used dynamic adaptation in a manner of *constrained adaptation* for the development of safe and reliable embedded systems for several years now. By constrained adaptation we refer to the fact, that we require complete specifications of the adaptation behavior already at development time, which are evaluated by the system at runtime in order to adapt to the current runtime situation. Since the models forming the basis for the adaptation behavior are available at design time, we are able to conduct thorough analysis of the specifications for the purpose of validation and verification. We are

able to guarantee deterministic adaptation behavior at runtime, which is obviously essential for safe and reliable systems [5][10].
We are convinced that the results and achievements in this domain can be of interest to the dynamic adaptation community in general. For this reason, this position paper points out some of the lessons we have learned over the last years of applying and advancing dynamic adaptation for the development of safe and reliable adaptive systems in industry and in research. Furthermore we figure out the major trends and research challenges that have to be addressed. To this end, section 2 provides an overview on the engineering of safe and reliable systems using dynamic adaptation and presents four typical stages of evolution of adaptive system development and classifies the current state of the art and state of the practice accordingly. Based on these observations we point out some major challenges that have – from our point of view – to be encountered in future research in section 3. We conclude our paper in section 4 with a short summary and an outlook on future work.

2. Dependable Adaptive Embedded Systems

Safety and reliability are typical and very important non-functional properties of embedded systems. Safety and reliability engineering (SRE) has therefore been one of the most important sub disciplines of embedded system development for several decades now. In general, the goal of SRE is to reduce the probability of system failures. Commonly accepted approaches of SRE are redundancy patterns (e.g. heterogeneous redundancy), error handling (e.g. forward or backward recovery), and shut-down systems (e.g. safety-executive-pattern). In recent years, however, dynamic adaptation has emerged as an additional, very efficient technique to improve system safety and reliability. In the case of errors, the system adapts to compensate these errors as far as possible. In some cases it is accepted that an error leads to a degraded functionality as long as the safety of the system is ensured (graceful degradation or survivability [18]).
Most embedded systems have to interact with highly dynamic environments. Dynamic adaptation has therefore always been an inherent key element of such systems. In order to apply dynamic adaptation for the development of safety-critical and reliable systems, it is necessary that the adaptation behavior is explicitly defined and that the negative as well as positive impact on safety and reliability can be measured. Since the specification of the adaptation behavior is a complex and error prone task, a systematic software engineering approach for the development of such systems is required. However, such constructive methodological support for the development of safe and reliable embedded systems is still in its infancy. To regard this aspect more systematically, we have assigned our observations on applying adaptation in research and industry to four evolution stages:

- **Stage 0: non-adaptive systems**
  In this stage, the system realizes no kind of dynamic adaptation. This applies only to those embedded systems that do not (need to) adapt to any kind of environmental changes.

- **Stage 1: implicit adaptation**
  Most embedded systems are at least at evolution stage one. At this stage, the adaptation behavior is modeled as indistinguishable part of the functionality. We would call any system at this evolution stage or beyond an adaptive system. The motivation to use dynamic adaptation at this stage is mainly the necessity to adapt to dynamic environments. If we regard a vehicle stability controller, it is necessary to estimate the current driving situation. Decisions and control strategies then depend on this context information. We call this implicit dynamic adaptation, since there is definitely an adaptation although most developers do neither know that they currently develop an adaptive system nor that they have an idea of the implications of dynamic adaptation. Since the adaptation behavior is not explicitly modeled, adaptations often happen locally at a component level. The dependencies between different components cannot be captured and are often not considered at all. This leads to serious problems since adaptations in one component usually have an influence on the quality of the provided services of the component. Not communicating this influence to

relying components often leads to serious failures. The latter are difficult to reconstruct and it is hardly possible to identify the causing faults.

- **Stage 2: explicit adaptation, no engineering of adaptation**
  Starting at stage 2, dynamic adaptation is explicitly considered in system development. Most of the research of recent years has been focused on this stage. Also in industry some systems have already reached this stage. The main characteristic that makes a system belonging to this stage is the presence of a dedicated runtime adaptation framework. This framework could be a central component in the system coordinating all adaptation processes or it could be a decentralized aspect that is scattered to different components. In any case, however, the dynamic adaptation is explicitly controlled and/or coordinated. For industry, the main reason to evolve into this stage is the system quality. Some companies already noticed that implicitly used dynamic adaptation is a major cause for the troubles they have. The adaptation frameworks are usually quite simple and require a model or specification telling them under which condition which adaptation strategy has to be chosen. For complex systems it is hardly possible to define such a specification ad hoc without applying an appropriate, constructive development methodology. Therefore this leads to another challenge. The complexity of dynamic adaptation that has been neglected at stage 1 is now made visible. Although the quality problem can be encountered, an immense effort is required to manage the complexity of the adaptation behavior.

- **Stage 3: software engineering of adaptive embedded systems**
  This constitutes the currently final stage. In this stage not only an execution platform or mechanism to realize dynamic adaptation at runtime is provided, but also a dedicated methodology enabling developers to systematically develop adaptive embedded systems. First, this includes a seamless modeling methodology. In this regard, it is important to make the complexity manageable, e.g. by supporting the modular and hierarchical definition of adaptation. Second, the seamless software engineering approach also includes the model-based analysis, validation and verification of dynamic adaptation. For dependable embedded systems, it is indispensable to have a means to analyze the adaptation behavior already at design time and to guarantee certain properties. Therewith this model-driven approach makes it possible to identify reasonable configurations in an early stage of the development process – without first implementing them. Furthermore, this stage obviously also benefits from the whole range of typical gains brought by model-driven engineering (MDE) [19] approaches (i.e. validation, verification, reuse, automation) As for any other software engineering approach it is particularly possible to analyze and to predict the quality of the adaptation behavior to enable systematic control of the development process.

If we come back to the development of safe and reliable systems, it seems to be a self-evident conclusion to apply dynamic adaptation. The available concepts in industry, however, are mostly not sufficient. In fact, the development of such systems requires systematic software engineering approaches, i.e. it is essential to aim at reaching evolution stage 3. For this reason, it is in our opinion one of the most important research challenges to provide appropriate software engineering approaches for the development of adaptive (embedded) systems.

Research activities started at evolution stage 2 several years ago. Thus a vast majority of the researches concerning constraint adaptation are dealing with the development of runtime adaptation frameworks ([3], [6] [12], [13] to name a few examples).
A project of Carnegie Mellon University called RoSES [6] uses product family architectures for realizing dynamic reconfiguration. They define a product family and, in the case of faults they reconfigure the system to an alternative product configuration from this product family.
[3] uses a *low-fidelity* high-speed search algorithm and a *high-fidelity* search algorithm to determine the next system configuration. If a reconfiguration is subject of hard timing constraints, e.g., if failures occur that affect critical system services, the high speed algorithm searches for a viable

configuration that implements all critical functions. The *high-fidelity* algorithm that searches high-utility system configurations is applied when no timing constraints are given, e.g., a viable configuration is currently active.
In [13] so-called Containment Units monitor the quality of functionalities. Depending on the detected quality, the containment unit turns the functionalities off or replaces them with alternative functionalities.
The researches of the embedded adaptive systems laboratory EASL [14] deals with the transitions between configurations, taking into account that the configurations might have continuous or discrete states. Although the EASL does not aim at the development of a framework their research results contribute to evolution stage 2, because they take the specification of the adaptation behavior for granted and focus on the realization of the adaptation behavior.

Since state of the practice has reached stage 2 and many researches have shown that the realization of the adaptation behavior based on a given specification is manageable, the importance of evolution stage 3 has most recently increased in research and many approaches for quality assurance of adaptive systems emerged.
In [1], the authors introduce a method for constructing and verifying adaptation models using Petri nets. In [2], linear temporal logic is extended with an 'adapt' operator for specifying requirements on the system before, during and after adaptation. An approach to ensure correctness of component-based adaptation was presented in [4], where theorem proving techniques are used to show that a program is always in a correct state in terms of invariants. [15] introduces a formal model of reconfiguration and an associated set of high-level, general system dependability properties that can be verified.
Although these approaches already support verification of dynamic adaptation, the current state of the research is at the very beginning of stage 3. The main reason for this is that they are based on adaptation behavior specifications without providing adequate constructive modeling methodologies. Therefore these specifications are hardly to define in a reasonable manner for real systems. For instance specifying adaptation behavior using Petri nets [1] is not an intuitive way to design complex industry sized systems like the ESP (Electronic Stability Program). Furthermore, in current researches the quality assessment of adaptive systems is solely based on verification techniques, however, other techniques like probabilistic analysis are indispensable for quality assurance in particular with respect to assurance of safety requirements.

As an example our MARS project aims at providing a seamless engineering approach from the requirements to running systems. MARS uses dynamic adaptation as a flexible error handling technique aiming at cost-efficient development of dependable embedded systems. Starting from a Feature-Model [16] the system architecture is defined using the Mars modeling language [5], which is basically an extension of established concepts of architecture description languages [17]. For this purpose we use the modeling environment GME [8].
From the analysis model that specifies the adaptation behavior, a design model in Matlab/Simulink™ is generated that combines adaptation and functionality in an integrated model building the basis for the subsequent system design [11]. The validation and verification techniques of the adaptation behavior include simulation, verification [9], and probabilistic analyses [10].

3. Future Challenges

For the development of safe and reliable embedded systems based on dynamic adaptation, it is indispensable to come to a seamless software engineering approach. One aspect is definitely an integrated methodology guiding the developer systematically from the requirements to a validated and verified adaptive system. As for any other software engineering approach, it is furthermore a key concern to answer the question what the measurable benefits of using dynamic adaptation are. In the given context this means, that it is indispensable to come to a possibility to measure the impact of dynamic adaptation on safety and reliability. Otherwise it is not possible to evaluate and to compare different approaches. Neither it is possible to evaluate the chosen adaptation strategy and thus to

control and to steer the development process. For example, it is some cases reasonable to use dynamic adaptation to realize a full-fledged redundancy, in other cases it might realize a simple shut-down system, and in some cases the system might be highly adaptive realizing various different degradation levels. A systematic decision which variant is most appropriate in a given context requires a possibility to measure the effects on safety and reliability and to relate these effects to the requirements and the costs.
For this reason, it is important to come to a common understanding of safety and reliability of adaptive systems and to provide a means of measuring these values.

4. Conclusion and Future Work

Typically there is a broad range of different drivers for dynamic adaptation in embedded systems. We focus on the application of dynamic adaptation for the development of safe and reliable systems, since dynamic adaptation is a feasible approach which provides an immense potential for saving hardware costs, for ensuring the safety of the system, and for achieving acceptable availability. As described in the stages of evolution of adaptive systems, a fundamental step forward to improve the system quality can be achieved by explicitly considering the adaptation behavior. However, in order to assure the specified safety and reliability level of a system and also minimize its related hardware costs, a seamless engineering approach including quality and safety prediction is indispensable.
Although our MARS project provides a constructive modeling methodology for specifying the adaptation behavior and several validation and verification techniques, there remains a lot of future work. In a first step we will try to quantify the safety and the reliability that comes from our adaptation behavior model. Obviously, this presumes a clear definition of the semantics of these attributes in an adaptive system which currently does not exists, e.g., there is no definition that defines how reliable a functionality is which is implemented in a gracefully degrading manner. Besides the determination of safety and reliability, we also intend to measure the hardware costs that are related to an adaptation model. In a second step, we would like to come to an approach to automatically find weak points in our adaptation model by applying typical safety or reliability pattern. Our according long-term objective is to automatically or semi-automatically improve a given MARS adaptation model which means to find a less expansive solution that still fulfills all requirements.

5. References

[1] J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering (ICSE'06)*, pages 371–380, Shanghai, China, 2006. ACM.

[2] J. Zhang and B.H.C. Cheng. Specifying adaptation semantics. *In Workshop on Architecting Dependable Systems (WADS'05)*, pages 1–7, St. Louis, USA, 2005. ACM.

[3] O. A. Rawashdeh and Jr. J. E. Lumpp. A technique for specifying dynamically reconfigurable embedded systems. *In IEEE Conference Aerospace*, 2005.

[4] S.S. Kulkarni and K.N. Biyani. Correctness of component-based adaptation. *In Symposium on Component Based Software Engineering (CBSE'04)*, volume 3054 of LNCS, pages 48–58, Edinburgh, Scotland, 2004.

[5] M. Trapp, R. Adler, M. Förster, and J. Junger. Runtime adaptation in safety-critical automotive systems. *In IASTED International Conference on Software Engineering (SE'07)*, Innsbruck, Austria, 2007. ACTA.

[6] RoSES: Robust Self-configuring Embedded Systems, URL:http://www.ece.cmu.edu/~koopman/roses/.

[7] Charles P. Shelton, Philip Koopman, William Nace, A Framework for Scalable Analysis and Design of System-wide Graceful Degradation in Distributed Embedded Systems, *Workshop on Reliability in Embedded Systems*, October 2001, New Orleans, LA. 2003.

[8] GME, http://www.isis.vanderbilt.edu/projects/gme/

[9] K. Schneider, T. Schuele, and M. Trapp. Verifying the adaptation behavior of embedded systems. *In Software Engineering for Adaptive and Self-Managing Systems (SEAMS'06), pages 16–22, Shanghai, China*, 2006.

[10] R. Adler, M. Förster, and M. Trapp. Determining configuration probabilities of safety-critical adaptive systems. *In IEEE International Symposium on Ubisafe Computing (UbiSafe'07)*, Niagara Falls, Canada, 2007. IEEE Computer Society.

[11] Andreas Beicht, Entwicklung eines Frameworks zur Entwicklung und Analyse adaptiver eingebetteter Systeme, Diplomarbeit, TU Kaiserslautern, Germany, 2007. In German.

[12] Depaude-Project Webpage. Dependability for embedded automation systems in dynamic environment with intra-site and inter-site distribution aspects URL:http://www.esat.kuleuven.be/electa/depaude/.

[13] Jamieson M. Cobleigh, Leon J Osterweil, Alexander Wise, Barbara Lerner, Containment Units: A Hierarchically Composable Architecture for Adaptive Systems, *Proceedings of SIGSOFT* FSE - 10, Charleston, USA, 2002.

[14] Wills, L.M., Kannan, S., Sander, S., Guler, M., Heck, B., Prasad, J.V.R., Schrage, D., Vachtsevanos, G., A Prototype Open Control Platform for Reconfigurable Control Systems, *Software-Enabled Control: Information Technologies for Dynamical Systems*, Piscataway, May 2003, pp. 63-84.

[15] Elisabeth A. Strunk, Reconfiguration Assurance in Embedded System Software, *PhD thesis, University of Virginia*

[16] M. Trapp, Modeling the Adaptation Behavior of Adaptive Embedded Systems. *PhD thesis, Technical University of Kaiserslautern,* 2005.

[17] N. Medvidovic and R. N. Taylor., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering, 26*(1), 2000.

[18] Knight, John C. and Elisabeth A. Strunk, Achieving Critical System Survivability through Software Architectures, *Architecting Dependable Systems*, 2004

[19] D.C. Schmidt, Model-Driven Engineering. *IEEE Computer* **39** (2). May 2006.

Applying Architectural Constraints in the Modeling of Self-adaptive Component-based Applications

Mohammad Ullah Khan[1], Roland Reichle[1], Kurt Geihs[1],

[1] University of Kassel,
34121, Kassel, Germany
{khan, reichle, geihs}@vs.uni-kassel.de

Abstract. In component-based software development, the architecture of a software system is represented as a composition of different connected components. Components can be atomic as well as composed of other components. In order to support application variability for potential runtime adaptations, we have defined a component framework, which introduces variation points by allowing alternative realizations for the application's components. Application variants as a basis for the adaptation decision are created at run-time by resolving the variation points, i. e. choosing a realization for each of the components. This may result in a large combinatorial number of alternative configurations. However, many of these application variants are usually not feasible, as some component realizations may require or exclude realizations for other components. The specification of appropriate architectural constraints is therefore inevitable to ensure feasibility of application variants and thus to help addressing the scalability problem. Because of the complex inter-dependencies among different aspects of the components, specifying such constraints can be a challenging task. In this paper, we present a solution to the definition of architectural constraints in the modeling of variability in component-based software development. The approach is applied in an MDA-based development process.

Keywords: Model Driven Development, Architectural constrains, Variability, Component-based software development, Self-adaptive applications

1 Introduction and Problem Statement

The use of handheld mobile devices is increasing quite rapidly. People are getting used to carrying some sort of mobile device like a PDA, smart phone or a laptop wherever they go. These devices usually have limited resources like battery power, memory, CPU capacity etc. and they are very often operating in vastly diverse and changing environments. Therefore, the performance and quality of applications running on those devices crucially depend on the resource constraints and the dynamically changing properties of the execution context, e.g. communication bandwidth fluctuates, error rate changes, battery capacity decreases, and a noisy environment may obliterate the effect of sound output. In order to provide a satisfactory output to the user, applications on mobile devices need to adapt

themselves to their current operational context automatically according to goals and policies specified by the user and/or the developer.

Our overall goal is to facilitate the development and operation of self-adaptive applications. The approach is based on dynamic compositional adaptation. We assume that applications are component-based. Context dependencies and variability of an application is specified as part of the application architecture. An application component can be hierarchically decom¬posed into other components. Each component may have a number of different realizations that provide the same basic functionality, but differ in their extra-functional characteristics like resource requirements and context dependencies. Therefore, variation points in the architecture of an application are introduced by the architectural options of choosing from a set of different realizations for a particular component. At application runtime, when there is a context change, all potential application variants are evaluated by the adaptation manager in the middleware by resolving the variation points. Note that this approach supports unanticipated adaptation insofar as new component realizations may be added at runtime to the component framework, thus effectively enlarging the number of variants dynamically.

In accordance with the MDA-based development process [1], the application adaptation model (in UML 2.0) is transformed by means of a model-to-text transformation into code that the adaptation middleware uses at runtime to resolve the variation points and to evaluate the utility of possible variants. If there is a context change during application execution, the underlying adaptation middleware computes on-the-fly all possible application variants and evaluates their fitness in respect to the current context situation. The “best” variant is selected and instantiated [2].

Resolving all the variation points with all the possible options can effectively create a huge number of different application variants, all of which have to be evaluated for fitness in the given context condition in order to find the best one. However, very often not all computed variants are actually feasible. Often, for example, the selection of a particular realization for a component implies a certain realization for another component. Likewise, a particular realization may be incompatible with some other component realizations. In order to ensure appropriate application variants for all context situations, infeasible variants have to be filtered out. Furthermore, the potential combinatorial explosion of variants can lead to a scalability problem requiring too much computational effort for a resource-scarce mobile device [3]. Therefore, filtering out the infeasible combinations drastically reduces the number of variants to be considered, and this comes as a very good answer to the scalability challenge.

For these reasons, we need a way to specify infeasible configurations but making the application modeling not overly complex. Hence, we have adopted an approach that builds on architectural constraints specified as part of the application model. The model is transformed into an internal representation that enables the adaptation middleware to filter out the infeasible combinations of components quickly at runtime. Thus, the resolution of the adaptation variation points is made faster substantially.

In the following, we present our approach for modeling architectural constraints. Our goals were to introduce a minimum of additional complexity in the model, and at

the same time to enable the specification of constraints among components appearing at any level of the hierarchical composition of an application.

2 Proposed Modeling Approach

We assume that component realizations may have some characteristic properties or features that have to correspond to the features of other component realizations or exclude some components realizations. In order to model these characteristics, we based our solution on a feature model such as described in [4]. In our approach, the features are associated with all the components (variation points) that are influenced by the feature. In addition, each of the features is associated with a simple constraint. These invariants, e.g., indicating if a feature should be common to all involved components or exclusively provided by only one component, are checked by the adaptation management when creating the application variants.

The feature model can be considered as a kind of crosscutting aspect with regard to the architectural model. Hence, our solution is a combination of concepts of feature models[4], invariants[5] and Aspect-Oriented Programming[6].

In order to explain our approach in detail, in the following we present a step-by step example with the help of UML models. The application, called "SatMotion" here, is one of the commercial applications, for which we have used our approach. For this paper, consider the names of the application and its components as arbitrary.

2.1 Identify the features

In order to identify the features, one has to closely analyze the application architecture, the component framework, the available components and their different variants. Then their inter-dependencies with regard to the resolution of variation points are identified and considered as features.

Figure 1 shows a composite structure diagram for the SatMotion application. The composition consists of four components (marked with the <<mComponent>> stereotype): *Controller*, *MathProcessor*, *Recorder* and *UserInterface*. Each of the components may have a number of different variants or realization options. For example, the *UserInterface* has variants called *OneWay, TwoWay* and *PlayBack* (see Figure 2). Although, each of these variants may as well have other different variants, and therefore each variant can introduce additional variation points, for simplicity we show only two decomposition levels; the same technique applies regardless of the number of levels in the variability hierarchy.

Now, *Controller* may also have similar variants and there may be the constraint that *'one way controller is compatible only with one way user interface'*. In addition, there might exist another constraint, e.g., *'a composition having a one way Controller can not have any variant of the MathProcessor'*. Thus, we identify two features: a feature called *'TypeMatching',* with its variants *'OneWayType', 'TwoWayType', 'PlayBackType', as well as a feature called 'CtlrMPIncompatibility', with its variant 'CtlrMPOWIncompatibility'*.

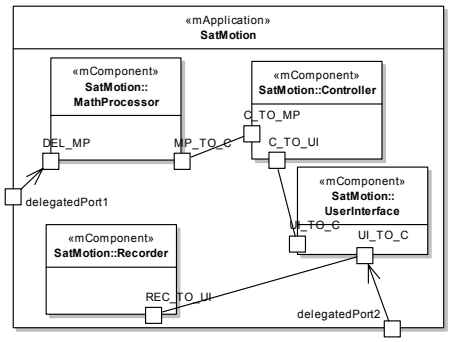


Figure 1: A composition of components for the SatMotion application

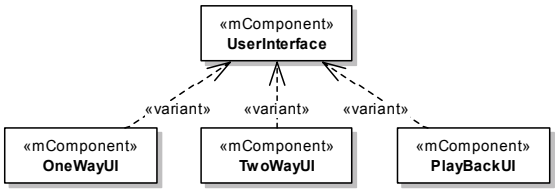


Figure 2: Different possible variants for the UserInterface component

2.2 Build a feature hierarchy

The next step is to build a hierarchy of the features. Just as the components of the architecture model, features can have a number of alternative realizations. In turn, each of the realizing features can be considered as abstract and can also have different realizations.

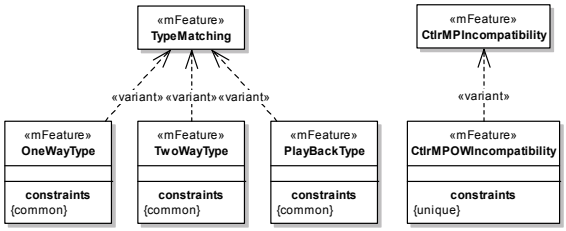


Figure 3: Part of a feature hierarchy

Therefore, the features, marked with the <<mFeature>> stereotype, are modeled in a hierarchical manner and we refer to the model as feature hierarchy. Furthermore, different hierarchy levels of the feature model and the architectural model are likely to

correspond, as features are associated to the components and the feature variants to the corresponding component realizations/variants.

For the simple example of subsection 2.1, we use only one level of abstraction and can thus form a feature hierarchy model as shown in Figure 3. In order to be able to specify that features of different components correspond, we provide the 'common' constraint in our modeling approach. For specifying that a feature of one component realization excludes the same feature of a realization of other components, we have introduced the 'unique' constraint. Currently, only these two constraints are available, because they are supported by our adaptation middleware. However, the approach is open to be enhanced with any number of other constraints.

2.3 Associate features with components

The next step is to associate features with the components in the composite structure diagram. Thus the structure of Figure 1 enhanced with the two features will result in the structure diagram shown in Figure 4.

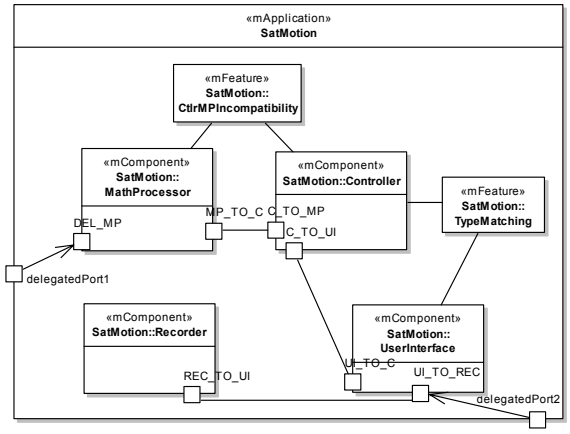


Figure 4: The application architecture of Fig. 1 is enhanced by the addition of features

This model indicates that UserInterface should consider the descendants (variants) of the 'TypeMatching' feature at their variation points and MathProcessor should be aware of the variants of 'CtlrMPIncompatibility' feature, while Controller is affected by both of these features.

With the variants of the components at this level, the corresponding variants of the features are associated. Some examples are shown in Figure 5.

Thus, when variation points are resolved for the model in Figure 4, the feature associations in Figure 5 dictate that if a *OneWayUI* is chosen for *UserInterface*, then for *Controller, OneWayController* must be chosen (constrained by '*common'*). On the other hand, if *OneWayController* is chosen for *Controller*, then neither *LSMathProcessor* nor *HSMathProcessor* can be chosen, as dictated by the *'unique'* constraint.

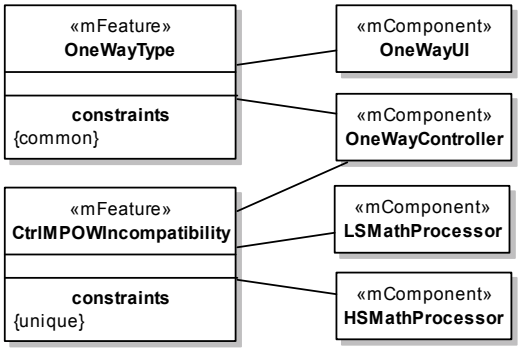


Figure 5: Association of features with the variants of the components used in Figure 4

The approach is very effective in quickly identifying and filtering out all the infeasible combinations. For this particular example, considering that each of the components in Figure 1 has three different variants, while applying the architectural constraints the proposed approach will produce only the feasible 21 variants. Thus, we can filter out 60 infeasible combinations out of the total 81 possibilities. Thus, during the fitness evaluation, the middleware can discard almost 74% infeasible variants resulting in a great improvement to the scalability problem.

3 Transformation and Deployment

We use an MDA-based development approach to support self-adaptiveness. One of the main mottos of the MDA-based development is to automate the generation of source code from the models. Among other advantages, it speeds up the development process and promotes error-free development. Therefore, as part of a comprehensive tool chain, we have built model transformations that transform the application model including the architectural constraints to appropriate source code which is deployed to our middleware supporting adaptive applications.

3.1 Transformation Support

We use the MOFScript model-to-text transformation tool[7] from the MODELWARE project[8] to transform the adaptation model written in UML 2.0 to Java source code. MOFScript comes as an Eclipse plug-in and therefore the whole transformation procedure is integrated within the Eclipse framework. The tool chain that is used for the modeling and transformation is presented in figure 6.

One requirement for the transformation using MOFScript is that the model must be expressed in a format supporting the Eclipse UML2 metamodel, which is a subset of the OMG UML 2.0 metamodel. In our work, we have used Enterprise Architect as the modeling tool. The result is a model in OMG UML 2.0. Therefore, an XSLT

stylesheet was also developed, which transforms the UML 2.0 model (exported XMI from Enterprise Architect) to UML2 (XMI) format, which can then be used by MOFScript as input.

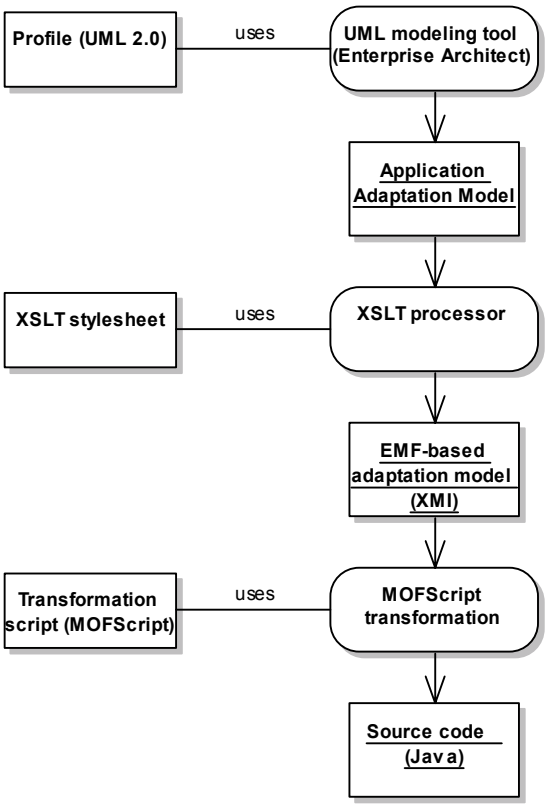


Figure 6: The tool chain for the modeling and transformation

In Figure 7 we show a code fragment that is generated for the architectural constraint part of the model.

```
HashMap featureSpec = new HashMap();
featureSpec.put("TypeMatching", new Feature("TypeMatching",
        "OneWayType", Feature.COMMON_CONSTRAINT));
BLUEPRINT_PLANS[11].setFeatureSpec(featureSpec);
```

Figure 7: Some code fragment, automatically generated from the model

Feature names 'TypeMatching' and 'OneWayType' are generated for the corresponding names in the model, while the COMMON_CONSTRAINT in the generated source code corresponds to the 'common' constraint, as introduced above. The middleware supporting the adaptive applications uses the generated source code for filtering out all infeasible combinations at runtime, based on the feature names and the constraints.

3.2 Deployment of the generated code

The generated source code (for the complete application) is then packaged in a jar archive and deployed to the middleware. In our work, we have successfully applied the approach to develop two adaptive pilot applications. The details of the middleware and the deployment of the source code are beyond the scope of this paper. Please refer to the (public) deliverables of the MADAM project[9].

4 Related Work

Specification of variability models has been addressed in many works over the years[10][11][12]. However, the focus of this paper is on the integration of architectural constraints in the variability model. The coverage of this in existing solutions is still quite limited, especially when adopting the MDA-approach of software development. For example, [10] addresses three types of dependencies constraining a variation point as well as the relationships among the dependencies. However, the dependencies must be modeled separately and thus applying the concept would produce a rather big overhead as far as the model is concerned. Therefore, integration of this concept in a model-driven development approach is quite cumbersome.

Different ADLs (Architectural Description Language) have been applied to specify architectural constraints. In ADLs like ACME[13], architectural constraints are expressed in a first-order predicate logic language. The scope of constraints can be component or subsystem-wide. In other ADLs like C2, where connectors are first class modeling elements, constraints can be placed in connectors, enforcing a set of policies in the components attached to it[14].

In [15], authors tackle the problem of software maintenance and reconfiguration, which often is costly due to the lack of documentation and automatic checking. Architectural invariants constrain the reconfiguration in order to maintain consistency. In a different track, work on the Fractal Component Model has spun off a number of subprojects, one of them being ConFract, a contract system for Fractal. Contracts, as defined by [16], capture - in the form of a predicate language - assumptions about the functional and extra-functional properties of components that must not be violated upon invoking an interface method.

A debate whether UML is a good ADL has been the topic of several papers and panels in the past years[17]. Although recent UML versions incorporate ADL concepts – like ports and connectors, the question whether UML and its OCL are adequate to express architectural constraints is still a topic needing further research.

The feature concept, used in the area of Software Systems Families, is defined in [4] as an important, distinguishable, user visible aspect, quality or characteristic of a software system. Features are organized in a hierarchical fashion to describe a system.

In our approach, we targeted the MDA-based development approach and thus UML comes more or less as an automatic choice for the modeling language. The feature hierarchy concept of [4] fits well with the component variability model, while crosscutting aspects of the components in the architecture can be considered as

features. The approach has its novelty in being able to evaluate features at runtime and thus facilitating adaptation which is unanticipated at design time. Whereas in the product line community an application variant is created based on pre-selected features, in our approach the features and the corresponding constraints are used to evaluate the feasibility of automatically created application variants at run-time.

5 Conclusions

We have presented an approach for modeling architectural constraints as part of a comprehensive model-driven development methodology for self-adaptive applications. It comes as a solution to filtering out infeasible component combinations in a variability model for component-based adaptive applications. The approach has been successfully applied in the development of two adaptive applications in the MADAM project[9].

From a modeling point of view, we introduce additional constructs in the form of a feature model. The feature model can be separately specified. It introduces a minimum of complexity in the architecture model of the application. From a performance point of view, the approach can very effectively filter out the infeasible application variants. Thus it provides an effective answer to the scalability question, which can be a threat to exploiting variability concepts in adaptive applications, more specifically for resource-scarce mobile devices.

In the current solution, we have used only two constraints *'common'* and *'unique'*, expressing the necessity and mutual exclusion of components (and their variants/realizations). The current middleware prototype (downloadable as open-source from [9]) supports only these two constraints. However, from the modeling point of view, the approach is applicable for any number of constraints of any form and thus it is easily extensible. In our future works, we will integrate OCL and other standard constraint specification languages into our approach. However, the challenge basically remains on the middleware, which must support such specifications.

References

1. Model Driven Architecture, http://www.omg.org/mda/
2. Kurt Geihs, Mohammad U. Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, "Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices". IFIP Working Conference on Software Engineering Techniques, October 17-20, 2006, Warsaw, Poland.
3. Mourad Alia, Geir Horn, Frank Eliassen, Mohammad Ullah Khan, Rolf Fricke and Roland Reichle, "A Component-based Planning Framework for Adaptive Systems". The 8th International Symposium on Distributed Objects and Applications (DOA), Oct 30 - Nov 1, 2006, Montpellier, France.
4. Streitferdt, D., et al. Configuring Embedded System Families Using Feature Models. In proceedings of 6th International Conference Net.Objectdays. 2005. Erfurt, Germany. p. 339-362.

5. Ahlgren, B., et al. Invariants: A new design methodology for network architectures. In proceedings of SIGCOMM 2004 Workshop on Future Directions in Network Architecture (FDNA'04). 2004. Portland, Oregon, USA. p. 65-70.
6. France, R., et al. An aspect-oriented approach to design modeling. IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2004.
7. MOFScript Model-to-Text transformation, http://www.eclipse.org/gmt/mofscript/
8. MODELWARE project homepage, http://www.modelware-ist.org/
9. MADAM project homepage, http://www.ist-madam.org/
10. Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. In Proceedings of the Third Software Product Line Conference (SPLC 2004) (Boston, MA, USA, August 30 - September 2, 2004). 197--213.
11. M. Clauss: Modeling variability with UML, GCSE 2001 - Young Researchers Workshop, September 2001.
12. S. Thiel, A. Hein: Systematic integration of Variability into Product Line Architecture Design, Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2), August 2002.
13. Garlan D., R.T. Monroe, D. Wile, Acme: Architectural Description of Component-Based Systems, in Foundations of Component-Based Systems, G.T. Leavens and M. Sitaraman, Editors. 2000, Cambridge University Press. p. 47-68
14. P. Oreizy, D. S. Rosenblum, and R. N. Taylor On the Role of Connectors in Modeling and Implementing Software Architectures, Feb 15, 1998 http://www.isr.uci.edu/architecture/papers/TR-UCI-ICS-98-04.pdf
15. T. V. Batista, A. Joolia, and G. Coulson, "Managing Dynamic Reconfiguration in Component-Based Systems", 2nd Int. Workshop on Software Architecture, vol. 3527 of LNCS, Springer, Pisa, Italy, p. 1-17, June, 2005.
16. P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre, "A Contracting System for Hierarchical Components", 8th Int. Symposium on Component- Based Software Engineering, vol. 3489 of LNCS, Springer, St. Louis, MO, USA, p. 187-202, May, 2005.
17. David Garlan and Andrew J. Kompanek Reconciling the Needs of Architectural Description with Object-Modeling Notations, Proceedings of the Third International Conference on the Unified Modeling Language October, 2000, York, UK

A Model-driven Approach to the Development of Autonomous Control Applications*

Helge Parzyjegla[1], Michael A. Jaeger[1], Gero Mühl[1], and Torben Weis[2]
{parzyjegla, michael.jaeger, g_muehl}@acm.org,
torben.weis@uni-due.de

[1] Communication and Operating Systems Group, Berlin University of Technology,
Einsteinufer 17, 10587 Berlin, Germany
[2] Distributed Systems Group, University Duisburg,
Bismarckstraße 90, 47057 Duisburg, Germany

Abstract. Actuator and sensor networks (AS-Nets) will become an integral part of our living and working environment. AS-Nets are formed by modern end-user devices (ranging from PCs over PDAs and TV/HiFi-systems to service robots) that communicate wirelessly (e.g., by using WLAN, Bluetooth, or IrDA) and may cooperatively provide services in e-Home scenarios. This paper presents a model-driven approach to the development of applications for AS-Nets that reliefs developers from worrying about heterogeneity, distribution, faults, and self-organization by encapsulating the necessary expert knowledge in the model transformation. Moreover, knowledge derived from the model is exploited at runtime to adapt the application to dynamic changes in the environment.

1 Introduction

Developing applications for actuator and sensor networks (AS-Nets) in an e-Home scenario is challenging. The heterogeneity of devices and networking technologies vastly increases the complexity. Moreover, frequent reconfigurations and communication faults (such as network partitioning) have to be tackled, too. Both issues state a problem for the developer since he can neither completely predetermine the configuration nor anticipate all potential faults that might occur at runtime. Furthermore, the user will probably not be willing or may even not be able to handle complex configuration issues or faults at install time or runtime. Hence, devices and applications must be able to work as autonomously as possible, requiring only little to no manual intervention.

In this paper, we present a model-driven approach to the development of autonomous control applications. The goal is to empower the application developer to create self-organizing and robust applications for AS-Nets with only minimal expert knowledge. Therefore, we do not only use the model for the design and deployment of applications but also to dynamically adapt them at runtime.

* The presented work is part of the *Model-Driven Development of Self-Organizing Control Applications* (MODOC) project that has been funded by the German Research Foundation (DFG) priority program 1183 *Organic Computing*.

The remainder of the paper is structured as follows. Section 2 presents the application model, the modeling language, and the model transformation. Furthermore, it introduces roles and self-stabilization as fundamental concepts for our approach. Section 3 describes how meta-information gained from the model can be leveraged at runtime. The paper closes with conclusions in Section 4.

2 Application Model and Model Transformation

To support the development of autonomous control applications we aim at providing a tool chain that comprises an easy-to-learn modeling language, a graphical development environment, and a model transformation process that encapsulates necessary expert knowledge to deal with heterogeneity and self-organization at design time and runtime. Starting point is the application model as created by the application developer. Therefore, we designed a graphical modeling language that is tailored to the development of pervasive applications while trying to keep the complexity on a moderate level. Every programming construct has a visual representation enabling even novice programmers to familiarize quickly [4]. The modeling language is built on concepts derived from the π-Calculus [2].

The provided model is subsequently transformed in multiple steps. In the first step, it is analyzed and split into roles that cooperatively realize the application's functionality. Each role presumes a set of capabilities that describes the minimum requirements for a device to be able to serve this role. Since roles do only address other roles and not a concrete node, they decouple the distributed application from the node or device it runs on.

A robust role assignment [3] is an important precondition for this role-based development. Therefore, roles are equipped with self-stabilizing algorithms (e.g., publish/subscribe messaging) that aid self-organization at runtime. These algorithms are taken from an algorithm toolbox that contains different self-stabilizing algorithms for many purposes in a parameterizable fashion. In our approach, self-stabilization [1] is a key concept to achieve robustness. A self-stabilizing application is guaranteed to recover from any transient fault within a bounded number of steps provided that no further fault occurs until the system is stable again. Transient faults include temporary network link failures resulting in message duplication, loss, corruption, or insertion, arbitrary sequences of process crashes with subsequent recoveries, and arbitrary perturbations of data structures.

Finally, code is generated for the diverse target platforms while it is ensured that every node is at least equipped with self-stabilizing algorithms for inter-role communication and dynamic role assignment.

3 Leveraging the Role Model at Runtime

Self-stabilization mechanisms have their relevance at runtime (e.g., to realize fault tolerance), but cannot work alone without knowledge about the roles of an application. This knowledge is derived from the intermediary role model generated in the first transformation step and kept as meta-information during the rest

of the transformation process. For example, devices compare their capabilities to the requirements of a particular role that are stored in the meta-information in order to know which roles they are able to perform. A previously chosen node acting as role coordinator is responsible for assigning roles to capable candidates. The coordinator must know which roles belong to a particular application. Only if at least one candidate exists for every required role, the coordinator performs the assignment in order to prevent applications running only partially.

The more details of the intermediary role model are preserved as meta-information that are available to the role coordinator and the devices, the more adequately the role assignment can be carried out at runtime. If the role requirements are additionally enriched with meta-information about the desired quality of service, devices are able to advertise how good they are in performing a particular role. Using these advertisements the role coordinator can subsequently determine the most convenient assignment for the user in the present context.

However, this assignment currently does not take network limitations into account, i.e., it can happen that two roles that have to communicate heavily are assigned to different nodes that are far away from each other causing a high network load. In order to circumvent this, we plan to mark roles that are presumed to have a high communication demand during the model transformation. Subsequently, a revised role assignment mechanism can assign marked roles to nodes that are closely located to each other.

4 Conclusions

In this paper, we presented a model-driven development approach for autonomous control applications that encapsulates necessary expert knowledge in the model transformation process in order to free application developers from having to care about distribution, heterogeneity, deployment, and self-organization. In order to facilitate applications to organize themselves we build on a flexible role concept and the usage of self-stabilizing algorithms. Knowledge about behavioral aspects of the application as specified in the application model is a valuable resource that can be preserved in meta-information attached to generated code to be exploited for self-organization at runtime. We reason that models are a necessary prerequisite to be able to automatically guide self-organization processes to their intended goals.

References

1. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
2. A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Bioconcur'04*. ENTCS, Aug. 2004.
3. T. Weis, H. Parzyjegla, M. A. Jaeger, and G. Mühl. Self-organizing and self-stabilizing role assignment in sensor/actuator networks. In *The 8th International Symposium on Distributed Objects and Applications (DOA 2006)*, 2006.
4. T. Weis, A. Ulbrich, and K. Geihs. Model metamorphosis. *IEEE Software*, 20(5):46–51, Sept./Oct. 2003.

On Run-time Behavioural Adaptation in Context-Aware Systems

Javier Cámara, Gwen Salaün, and Carlos Canal

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
{jcamara,salaun,canal}@lcc.uma.es

Abstract. When systems are built assembling preexisting software components, the composition process must be able to solve potential interoperability problems, adapting component interfaces. In the case of Context-Aware systems, which exploit context information (*e.g.,* user location, network resources, time, etc.), the execution conditions are likely to change at run-time, affecting component behaviour. This work presents an approach to the variable composition of possibly mismatching behavioural interfaces. Our approach enables composition at run-time, rather than generating a full adaptor off-line. Keeping *Separation of Concerns* in the specification of the mapping, and applying a composition process which is able to handle new context information and components as they join the system, we also lay down the foundations for the reconfigurability of the system.

1 Introduction

Context-Aware computing [5] studies the development of systems which exploit context information (*e.g.,* user location, network resources, time, etc.), which is of special relevance in mobile systems and pervarsive computing. At the same time, software systems engineering has evolved from the development of applications from scratch, to the paradigm known as *Component-Based Software Development* (CBSD) [12], where third-party, preexisting software components known as *Commercial-Off-The-Shelf* or COTS are selected and assembled in order to build fully working systems. Due to their black-box nature, these components are equipped with public interfaces which expose information required to access their functionality.

However, most of the time a COTS component cannot be directly reused *as is*, and requires adaptation in order to solve potential problems at the different interoperability levels (*i.e.,* signature, protocol, service and semantic) with the rest of the components in the system. The need to automate this adaptation process has driven the development of *Software Adaptation* [3], a discipline characterised by the modification or extension of component behaviour through the use of special components called *adaptors*, which are capable of enabling components with mismatching interfaces to interoperate. These are automatically built from an abstract description of how mismatch can be solved (*i.e.* adaptation *mapping*), based on the description of component interfaces. Specifically, this work is focused on the protocol or behavioural level.

In mobile and pervasive computing, the execution context of the system is likely to change at run-time (*e.g.,* time, user location). Hence, an appropriate adaptation of

the components must dynamically reflect these changes which might affect system behaviour. In this work we advocate for the use of dynamically modifiable adaptation policies between an arbitrary number of components which depend on the current state of the execution of the system, considering additional policies which depend exclusively on context changes (*i.e.,* context-triggered actions). Furthermore, our approach simplifies the complexity of mapping specification relying on *Separation of Concerns* [1], and avoids the costly off-line generation of adaptors, adapting components at run-time by means of a composition engine which manages communication dynamically within the system.

The rest of this paper is structured as follows: Section 2 presents a *Wireless Medical Information System* as a case study. Section 3 details the case study and points out the main issues related with composition and adaptation, illustrating them with our example. Section 4 describes our composition/adaptation model and sketches some general implementation questions. Finally, Section 5 draws up conclusions and further work.

2 Case Study: Wireless Medical Information System

In order to illustrate the different issues addressed in this paper, we describe a *Wireless Medical Information System* based on a real-world example, simplified for the sake of clarity. The system consists of a client-server application which systematically processes the clinical information related to patients in a medical institution. There is a central server with a DBMS installed which is queried remotely from PDAs. Handheld devices and server are connected through a wireless network setup.

The client PDA must be able to work with three user profiles which have different privileges: while **Staff** can access a restricted set of information (*e.g.,* administrative info for attendants), **Doctors** and **Nurses** can access also medical information, and prescribe specific treatments for any given patient in the case of doctors. When a nurse applies a treatment previously prescribed by a doctor on a specific patient, the actions and/or medicines administrated must be entered in the application (treatment logging).

It is important to maintain the application operative on the PDA continuously, hence a lightweight DBMS component has been incorporated to each PDA, enabling the user to work locally whenever the wireless signal is lost (local mode). Moreover, since the storage on a PDA is very limited, only treatment prescriptions and logging are stored in the local DBMS. Patient information is retrieved from Radio Frequency Identification (RFID) tags fixed on patient bracelets when in local mode. This is achieved through an RFID reader incorporated on each PDA. Every time the client on the PDA returns from local to remote mode, it is mandatory to synchronise the data stored locally with the central DBMS. This process must be automatically conducted by the application.

The client PDA is being reused from a legacy system which does not take into account user profiles, hence the appropriate restrictions must be applied at the composition level in order to limit the access rights to the DBMS as informally sketched above. Likewise, this client is built to work with a DBMS, independently of its location (the client does not know about the existence of the local DBMS nor the RFID reader), requiring adaptation to the new characteristics of the system.

3 General Issues

In order to describe the behaviour of components, we extend component interfaces with a description of the protocols they follow. Specifically we use *Labelled Transition Systems* (LTS), which are automatons taking the set of messages (both offered and required) in the signature of a component as input alphabet. Figure 1 depicts the different protocols for the components in our case study: The $CLIENT$ component can log an user in/out ($loginDoctor!/loginNurse!/logout!$), request the insertion of a given treatment on the database ($treat!$), log the administration of a treatment ($logtreat!$), or request some information to the server $query!$, returned by $response?$. It is worth noticing that the client grants the same privileges to all users. The $DBMS$ and $LOCAL\ DBMS$ components have analogous actions, with the exception that the latter only accepts $treat?/logtreat?$ requests, and $synch?$, which triggers a synchronisation process with the DBMS components (the details of this process are not described since they are not relevant for our example). Finally, the $RFID\ READER$ component first has to be enabled ($enable?$). Subsequently, it can receive a $read?$ command, returning the requested information on $data!$. In our case study, when the wireless signal is either found or lost, this will the represented by the pair of signals $connected!$ and $disconnected!$, respectively.

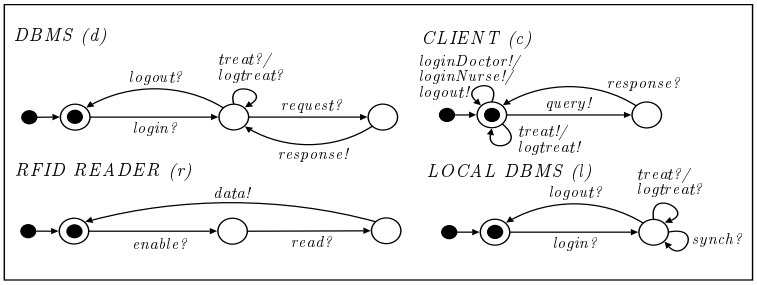


Fig. 1. Component protocols for the Wireless Medical Information System. Initial and final states are respectively noted in LTSs using bullet arrows and darkened states. Emissions and receptions are denoted by ! and ?, respectively.

Analysing our case study, we can highlight several issues which have to be carefully considered in the process of adaptation:

Complexity in specification. Supporting variable adaptation policies can lead to a remarkable degree of complexity in the specification of adaptation, especially in the case of many adaptation concerns. Considering every possible combination of context factors as individual states of the context is unfeasible in complex systems. Specifically, the composition in this system must take into account a couple of different concerns: **(i)** User profile. The client we are reusing does not distinguish user privileges, therefore we must provide the means to restrict user privileges based on user profiles (*e.g.*, a $treat!$ should not be issued to the DBMS unless a doctor user is logged in

-$loginDoctor$!-). **(ii)** Wireless coverage. Working in connected mode, queries are issued to the DBMS, but when in local mode, a $query$! request must be issued to the $RFID\ READER$ component.

Interoperability Issues. Independently of the different concerns to be considered for the composition, there are interoperability issues to be solved relative to the different interfaces of our case study: **(i)** Name mismatch occurs when a particular process is expecting a particular input event, and receives one with a different name (*e.g.*, $CLIENT$ sends $query$! while $DBMS$ is expecting $request$?). **(ii)** 1-to-many interaction is given if one or more events on a particular interface have not an equivalent in the counterpart's interface. If we take a closer look at the $CLIENT$ and $RFID\ READER$ interfaces, it can be observed that while the client is just sending $query$! when it wants to read some data, the reader is expecting two messages ($enable$? and $read$?). While the latter actually requests the data to the reader, the former has no correspondence on the $CLIENT$ interface. Hence, the composition process has to solve this mismatch by making the reader evolve independently, through the reception of $enable$? before each $read$? request.

Context-triggered actions. Different approaches to component behavioural adaptation [2, 13] deal exclusively with component communication. Hence, the adaptation process is driven by the exchange of component messages. However, there are actions required for a successful adaptation of components which depend on context state changes, rather than on component communication (*e.g.,* local-remote DBMS synchronisation in our case study). The notion of context-triggered actions is highly relevant in the field of Context-aware computing [11], although it remains obviated by the aforementioned proposals.

Reconfigurability of the system. Adaptor-based approaches to Context-aware adaptation [6] do need to consider every possible state of the system (not only context) during the adaptor generation process. This means that the adaptor is no longer valid if new context facets or components are added or removed at run-time, requiring the costly generation of a new adaptor.

4 Run-time Behavioural Adaptation

4.1 Composition/Adaptation Model

We propose a composition model which aims at tackling the different issues introduced in the previous section. This model includes the following elements:

(i) Interface Behavioural extensions, as described in Section 3. In addition, in order to describe changes in the state of the context, we also define an *Execution Environment* $E = \{e_1, \ldots, e_n\}$ as the set of events or signals which can be generated within the context of a particular system, and which do not belong to any particular component. As we have mentioned, in our case study, we will only consider the pair of environmental events $E =$\{connected!,disconnected!\} when the wireless signal is found or lost, respectively.

(ii) An expressive and intuitive **graphical notation** which reduces the complexity of describing mappings, through their incremental specification focusing on the different

concerns or *context facets* involved (separation of concerns). Each context facet is represented as an LTS, where the changes between its different states are triggered either by component messages or environmental events or signals. By keeping context facets separated we also reduce the computational cost of adaptation, since the number of context states to process experiments only a linear growth as context facets are added. Each context facet contains a set of *synchronisation vectors* [4] in order to denote communication between several components and the environment. A synchronisation vector (or vector for short) is a tuple where each event appearing is executed by a component or the environment, and the overall result corresponds to a synchronisation between all the involved components. Component messages or events are identified in a vector by prefixing their names with the component identifier, whereas environmental events are not prefixed, *e.g.,* $\langle c{:}treat!, l{:}treat? \rangle$, $\langle connected!, l{:}synch? \rangle$.

A vector may involve any number of components and does not require interactions on the same names of events as it is the case in process algebras [9, 8]. Vectors are associated to one or more states within the facet LTS, in such a way that it will only be active when the facet's current state is associated to the vector. In addition, the mapping may contain a set of vectors which are not associated to any particular context facet and are always active (global vectors). Facets have a precedence order assigned, hence the declaration of a vector in a facet with higher precedence overrides a lowest precedence declaration. Global vectors have a precedence order $p = 0$, and may be overriden by vectors on facets.

In order to illustrate how active vectors are selected in a given moment, we use a mapping for our case study depicted in Figure 2. We focus on the particular vector v_{ltr}, declared as $v_{ltr} = \langle c : logtreat!, d : logtreat? \rangle$ in the global set of vectors (the $logtreat!$ message is issued to the remote DBMS). v_{ltr} is defined as $v_{ltr} = \langle c : logtreat!, l : logtreat? \rangle$ in the $WIRELESS\ COVERAGE$ facet (the operation is performed on the local DBMS), and as $v_{ltr} = \langle c{:}logtreat! \rangle$ in $USER\ PROFILE$ (the operation is not performed, since $logtreat!$ corresponds to no action on the rest of the components). We also consider that the set of current states in facets are $C = \{DOCTOR, LOCAL\}$. Focusing on $WIRELESS\ COVERAGE$, we can observe that since v_{ltr} is associated to the $LOCAL$ state, the declaration on this facet overrides the global declaration. Similarly, since v_{ltr} is associated to $DOCTOR$ and the precedence of the $USER\_PROFILE$ is higher, the currently dominant declaration is again overriden. Finally, the operation is not performed since the prevailing declaration is $v_{ltr} = \langle c{:}logtreat! \rangle$. This is consistent with our example since doctors are not allowed to enter administrated treatments on the application.

Note that context-triggered actions are also represented in the mapping: after the local database has been updated ($DB\ UPDATED$ state in the $WIRELESS\ COVERAGE$ facet), when the wireless signal is recovered ($connected!$), the synchronisation of local and remote databases takes place. This is achieved by the definiton of the vector $v_{synch} = \langle connected!, l{:}synch? \rangle$.

(iii) From such an adaptation mapping, we propose a **composition process** to automatically compose and adapt a set of components at run-time. Figure 3 sketches the composition process: First, the set of active vectors dependent on the current states of the different facets of the context is selected. Second, run-time composition should

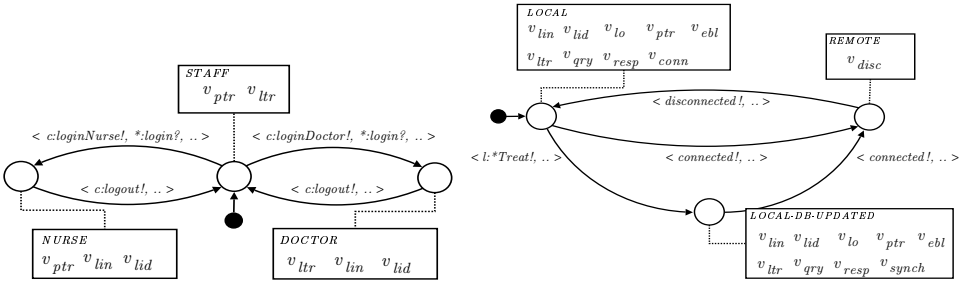


Fig. 2. Mapping facets for the case study: $USER\ \ PROFILE\ \ (UP)$ -left- and $WIRELESS\ COVERAGE\ (WC)$ -right-. The precedence order for the two context facets are $p(WC) = 1$ and $p(UP) = 2$, respectively.

avoid to engage into execution branches that may lead to deadlock situations. Considering the nature of the systems we are dealing with, we cannot ensure long-term correct termination of the system. On the contrary, the system in our case study is intended to operate continuously and its evolution may depend at some point on environmental factors which cannot be controlled. Since deadlocks cannot be statically removed (as it is done in approaches generating full adaptor descriptions), we have to ensure that each time a vector v is selected, there exists at least one global correct termination state for the currently running transaction after v is applied. Finally, once the vector v is processed, the state of components and context is appropriately updated.

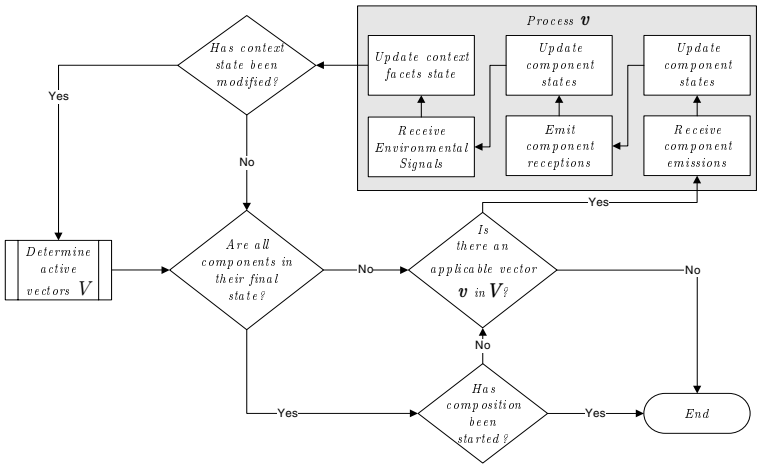


Fig. 3. Composition process.

4.2 Implementation Issues

We intend to implement this proposal as a composition engine, using *Aspect-Oriented Programming* (AOP) [7]. The main advantage is that unlike in traditional platforms and languages, a particular system can be modified without altering its code. This is achieved by separately specifying these modifications, and a description of their relation with the current system. Then the AOP environment relies on underlying mechanisms to *weave* or compose both original program and additional behaviour together into a coherent program. This weaving process can be performed at different stages of the development, ranging from compilation-time to run-time [10] (dynamic weaving). We are especially interested in this dynamic approach, where the virtual machine or interpreter running the code is aware of aspects and controls the weaving process. Hence, aspects can be applied and removed at run-time in a transparent way.

Dynamic AOP will enable us to shape up the composition engine as aspects able to: (i) intercept communication (*i.e.,* service invocations) between components; (ii) apply the composition process introduced in this proposal *wrt.* the adaptation mapping in order to make the right message substitutions; (iii) forward the substituted messages to their recipients transparently.

5 Conclusions and Open Issues

In this paper we have presented an approach to the composition of mismatching components in systems where its behaviour may be affected by the execution environment. Our approach applies composition at run-time rather than generating a full adaptor offline. This is achieved by means of applying separation of concerns to the specification of the mapping, reducing complexity in its specification, and a composition process which enables the addition and removal of new context information and components.

Regarding future work, our main perspective is to implement the whole proposal in a middleware using Dynamic AOP. In addition, while the nature of the mapping and the compositional process we have presented enables the transparent modification of the system, this work does not currently deal with the specifics of the reconfiguration process which takes place after the addition or removal of new context information or components as the system is running. Mapping or component update must be performed only at specific safe points, since the modification of this information at any other point could harm the correct execution of the system. The same applies to context changes during already running transactions, which should be able to execute correctly. A potential solution to this problem is delimiting the boundaries of transactions and delaying the application of context changes until they end.

Acknowledgements. This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

References

1. M. Aksit, J. Bézivin, and E. Roubtsova, editors. *Aspect-Based and Model-Based Separation of Concerns in Software Systems*, 2005.

2. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *The Journal of Systems and Software*, 74(1), 2005.
3. C. Canal, J.M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1), 2006. Special Issue on WCAT'04.
4. C. Canal, P. Poizat, and G Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *Lecture Notes in Computer Science*. Springer, 2006.
5. G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dartmouth College, 2000.
6. J. Cubo, G. Salaün, J. Cámara, C. Canal, and E. Pimentel. Context-Based Adaptation of Component Behavioural Interfaces. In *Proc. of COORDINATION'07*, LNCS. Springer, 2007. (in press).
7. R. Filman and D. P. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming Is Quantification and Obliviousness. Adisson-Wesley, 2005.
8. ISO. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation, 1989.
9. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
10. A. Frei Popovici A. and G. Alonso. A Proactive Middleware Platform for Mobile Computing. In *In Proc. of Middleware'03*, LNCS. Springer, 2003.
11. B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society Press, 1994.
12. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Adisson-Wesley, 2 edition, 2003.
13. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

Modeling Software Adaptation Patterns

Hassan Gomaa

Department of Information and Software Engineering
George Mason University
Fairfax, Virginia 22030, USA
hgomaa@gmu.edu

Abstract: This paper describes the concept of software adaptation patterns and how they can be used in software system adaptation and evolution. A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration.

1. Introduction

This paper describes the concept of software adaptation patterns and how they can be used in software system adaptation and evolution. Previous papers have described how software architectural patterns can be used to help in building software systems and product lines [Gomaa05a, Gomaa06]. This paper describes how software adaptation patterns can be used to help with the adaptation and evolution of software systems after original deployment. This paper describes different kinds of software adaptation. It then describes software architectural and design patterns and how they are used in evolutionary software design, before describing the characteristics of software adaptation patterns and how they are used in adapting and evolving software architectures.

2. Software Adaptation

Software adaptation addresses software systems that need to change their behavior during execution. In self-managed and self-healing systems, systems need to monitor the environment and adapt their behavior in response to changes in the environment [Kramer07]. Garlan [Garlan02] has proposed an adaptation framework for self-healing systems, which consists of monitoring, analysis/resolution, and adaptation. Kramer and Magee [Kramer90, Kramer98] have described how in an adaptive system, a component needs to transition from an active (operational state) to a quiescent (idle) state before it can be removed from a reconfiguration.

Adaptation can take many forms. It is possible to have a self-managed system which adapts the algorithm it executes based on changes it detects in the external

environment. If these algorithms are pre-defined, then the system is adaptive but the software structure and architecture is fixed. The situation is more complex if the adaptation necessitates changes to the software structure or architecture. In order to differentiate between these different types of adaptation, adaptations can be classified as follows within the context of distributed component-based software architectures:

a) Behavioral adaptation. The system dynamically changes its behavior within its existing structure. There is no change to the system structure or architecture.
b) Structural adaptation. Dynamic adaptation involves changing one component with another that has the same interface. The old component(s) has to be dynamically replaced by a new component(s) while the system is executing.
c) Architectural adaptation. The software architecture has to be modified as a result of the dynamic adaptation. Old component(s) must be dynamically replaced by new component(s) while the system is executing.

Model based adaptation can be used in each of the above forms of dynamic adaptation, although the adaptation challenge is likely to grow progressively from behavioral adaptation through architectural adaptation.

3. Software Architectural and Design Patterns

Software architectural patterns [Buschmann96, Gomaa05] provide the skeleton or template for the overall software architecture or high-level design of an application. These include such widely used architectures [Bass03] as client/server and layered architectures. Design patterns [Gamma95] address smaller reusable designs than architectural patterns, such as the structure of subsystems within a system. The description is in terms of communicating objects and classes customized to solve a general design problem in a particular context.

Basing a candidate software architecture on one or more software architectural patterns helps in designing the original architecture as well as evolving the architecture. This is because the adaptation and evolutionary properties of architectural patterns can also be studied and this assists with an architecture-centric evolution approach [Gomaa05b].

There are two main categories of software architectural patterns [Gomaa05]. Architectural structure patterns address the static structure of the software architecture. Architectural communication patterns address the message communication among distributed components of the software architecture.

Most software systems can be based on well understood overall software architectures. For example, the client/server software architecture is prevalent in many software applications. There is the basic client/server architecture, with one server and many clients. However, there are also many variations on this theme, such as multiple client / multiple server architectures and brokered client/server architectures. Furthermore, with a client/server pattern, the server can evolve by adding new services, which are discovered and invoked by clients. New clients can be added that discover services provided by one or more servers.

Many real-time systems [Gomaa00] provide overall control of the environment by providing either centralized control, decentralized control, or hierarchical control. Each of these control approaches can be modeled using a software architectural pattern. In a centralized control pattern, there is one control component, which executes a state machine. It receives sensor input from input components and controls the external environment via output components. In a centralized control pattern, evolution takes the form of adding or modifying input and/or output components that interact with the control component, which executes a state machine. Another architectural pattern that is worth considering because of its desirable properties is the layered architecture. A layered architectural pattern allows for ease of extension and contraction [Parnas79] because components can be added to or removed from higher layers, which use the services provided by components at lower layers of the architecture.

In addition to the above architectural structure patterns, certain architectural communication patterns [Gomaa05] also encourage adaptation and evolution. In software architectures, it is often desirable to decouple components. The Broker, Discovery, and Subscription/Notification patterns encourage such decoupling. With the broker patterns, servers register with brokers, and clients can then discover new servers. Thus a software system can evolve with the addition of new clients and servers. A new version of a server can replace an older version and register itself with the broker. Clients communicating via the broker would automatically be connected to the new version of the server. The Subscription/Notification pattern also decouples the original sender of the message from the recipients of the message.

4. Software Adaptation Patterns

The software architecture is composed of distributed software architectural patterns, such as client/server, master/slave, and distributed control patterns, which describe the software components that constitute the pattern and their interconnections. For each of these architectural patterns, there is a corresponding software adaptation pattern, which models how the software components and interconnections can be changed under predefined circumstances, such as replacing one client with another in a client/server pattern, inserting a control component between two other control components in a distributed control pattern, etc.

A software adaptation pattern defines how a set of components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of reconfiguration commands. A software adaptation pattern requires state- and scenario-based reconfiguration behavior models to provide for a systematic design approach. The adaptation patterns are described in UML with adaptation integration models (using communication or sequence diagrams) and adaptation state machine models [Gomaa04, Gomaa06]. An adaptation state machine defines the sequence of states a component goes through during from a normal operational state to a quiescent state, as shown in Figure 1. Once quiescent, the component is idle and can be removed from the configuration, so that it can be replaced with a different version of the component.

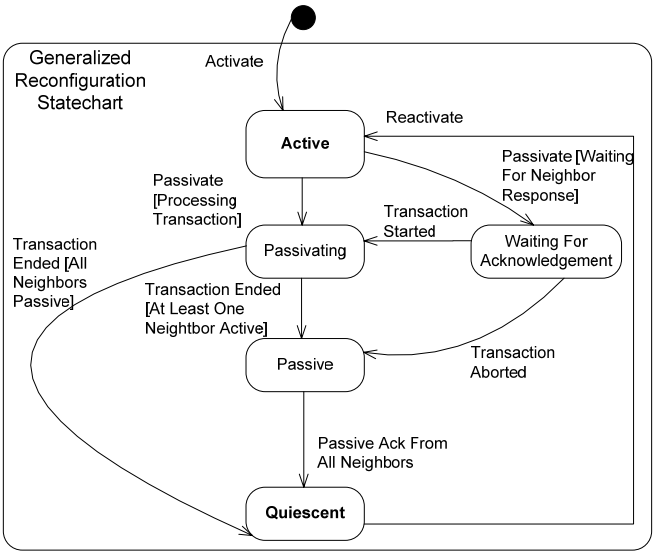


Fig. 1 Adaptation State Machine Model

5. Examples of Software Adaptation Patterns

Several adaptation patterns have been developed and are described below. These patterns can be used for all three kinds of software adaptation described in Section 2.

a) The Master-Slave Adaptation Pattern is based on the Master-Slave pattern [Buschmann96]. A Master component, which sends commands to slaves and then combines responses, can be removed or replaced from the configuration after the responses from all slave components have been received. Slave components can be removed or replaced after Master is quiescent.

b) The Centralized Control Adaptation Pattern is based on the Centralized Control pattern, and can be used in real-time control applications [Gomaa00]. The removal or replacement of any component in the configuration requires the Central Controller to be quiescent.

c) The Client / Server Adaptation Pattern is based on the Client / Server pattern [Gomaa05]. A client can be added to or removed from the configuration after completing the service request it initiated. A Server can be removed or replaced after completing the current service request.

d) The Decentralized Control Adaptation Pattern is based on the Decentralized Control pattern and can be used in distributed control applications [Gomaa00]. A control component in this Adaptation Pattern notifies its neighboring control components if it plans to become quiescent. The neighboring components cease to communicate with this component but can continue with other processing. Figure 1 shows the state machine model for a decentralized control component as it transitions from Active state to Quiescent state.

6. Conclusions

This paper has described the concept of software adaptation patterns and how they can be used in software system adaptation and evolution. A software adaptation pattern defines how a set of components that make up an architectural or design pattern dynamically cooperate to change the software configuration to a new configuration. For each software architectural or design pattern, there is a corresponding software adaptation pattern, which models how the software components and interconnections can be changed. This paper has outlined four software adaptation patterns. Several other adaptation patterns could be developed.

Further research includes effective approaches for automatically evolving software architectures using software adaptation patterns, in particular how to automatically select the appropriate adaptation pattern(s) to use, how to maintain a partial service while adaptation is taking place, and Quality of Service issues during software adaptation.

7. References

[Bass03] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison Wesley, Reading MA, Second edition, 2003.

[Buschmann96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, "Pattern Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

[Gamma95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Garlan02] D. Garlan and B. Schmerl, "Model-based Adaptation for Self-Healing Systems", Proc. Workshop on Self-Healing Systems, ACM Press, Charleston, SC, 2002.

[Gomaa00] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley, Reading MA, 2000.

[Gomaa04] H. Gomaa and M. Hussein, "Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures", Proc. Fourth Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, June, 2004.

[Gomaa05] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures, Addison-Wesley, 2005.

[Gomaa05a] H. Gomaa, "Building Software Systems and Product Lines from Software Architectural Patterns", ECOOP Workshop on Building Systems from Patterns, Glasgow, UK, July 2005.

[Gomaa05b] H. Gomaa, "Architecture-Centric Evolution in Software Product Lines", ECOOP Workshop on Architecture-Centric Evolution, Glasgow, UK, July 2005.

[Gomaa06] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Keynote paper, Proc. 9th Intl. Conference on Model-Driven Engineering, Languages, and Systems, Genova, Italy, Oct. 2006.

[Kramer90] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Engineering, Vol. 16, No. 11, Nov. 1990.

[Kramer98] Kramer J. and Magee J., "Analyzing Dynamic Change in Software Architectures: A Case Study", IEEE Int. Conf on Configurable Distributed Systems, Annapolis, May 1998.

[Kramer07] Kramer J. and Magee J., "Self-Managed Systems: an Architectural Challenge", Proc Intl. Conference on Software Engineering, Minneapolis, MN, May 2007

[Parnas79] Parnas D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments

Jérémy Dubus[1] and Philippe Merle[2]

[1] **Laboratoire d'Informatique Fondamentale de Lille - UMR CNRS 8022**
GOAL / INRIA ADAM Team
Université des Sciences et Technologies de Lille - Cité Scientifique
59655 Villeneuve d'Ascq CEDEX— FRANCE
Email: Jeremy.Dubus@lifl.fr
[2] **INRIA - ADAM Team**
INRIA FUTURS Parc Scientifique de la Haute Borne
Avenue Halley B.P. 70478
59658 Villeneuve d'Ascq CEDEX — FRANCE
Email : Philippe.Merle@inria.fr

Abstract. New distributed systems are running onto fluctuating environments (*e.g.* ambient or grid computing). These fluctuations must be taken into account when deploying these systems. *Autonomic computing* aims at realizing programs that implement self-adaptation behaviour. Unfortunately in practice, these programs are not often statically validated, and their execution can lead to emergent undesirable behaviour. In this paper, we argue that static validation is mandatory for large autonomic distributed systems. We identify two kinds of validation that are relevant and crucial when deploying such systems. These validations affect the deployment procedures of software composing a system, as well as the autonomic policies of this system. Using our DACAR model-based framework for deploying autonomic software distributed architectures, we show how we tackle the problem of static validation of autonomic distributed systems.

1 Introduction

The nature of the networks used to deploy distributed systems is changing. Well-defined networks, with well-known hosts, are no longer employed. The new emerging environments are *Open Dynamic Distributed Environments* (ODDE). Ambient, grid or sensor networks are the most known of these ODDE. In such environments, hosts can appear or disappear at any time. These changes in the environment have an impact on the applications deployed, hence these fluctuations must be taken into account to adapt properly their software architectures.

Autonomic computing [1] proposes to solve the problem of software self-adaptation, by introducing the concept of *autonomic policies*, which are the entities in charge of ensuring the adequate runtime reconfiguration of the system. The whole set of autonomic policies defined for a system represents what we call the *autonomic behaviour* of this system. Unfortunately, all environments that have emerged from this paradigm propose to write autonomic policies in a programmatic way, or using reconfiguration scripts. Programs or scripts that implement autonomic behaviour of a system only enable syntactical or semantical verifications. But we argue that these verifications are not sufficient. Indeed our opinion is that behaviour verifications are also needed to ensure that the system will not reach unconsistent behaviour state in some cases. We have identified two kinds of validations that support our position.

First, long-life systems that are modified very often during execution: software are installed then uninstalled, and this is repeated several times. In such a context, we have to ensure that every action performed in some installation process is undone in the opposite uninstallation process. Otherwise, undesirable side effects (*e.g.* a process started in the installation procedure that is not killed in the uninstallation procedure)

can occur and grow during the lifecycle of the system. These side effects can somehow lead to a crash of the system.

Second, autonomic policies can also interfere with each other and produce unexpected emergent behaviour (*e.g.* infinite loops). The problem has already been partially identified in the domain of active databases and is known as the *Feature Interaction* problem [2].

In this paper, we also introduce our proposition to solve the issue of validation of large and complete software autonomic systems in ODDE. This proposition relies on our DACAR model-based framework for building autonomous architectures [3, 4]. In this paper, we propose to extend the DACAR metamodel in order to validate autonomic policies. Two validations must be performed : First every action performed when a software is installed/started must be cancelled when this software is uninstalled/stopped. Second, the autonomic policies must be introspected to detect feature interactions such as cycles possible in the policy stack execution.

The remainder of this paper is the following. Section 2 presents the key research challenges of this work. Section 3 explains how we extend our DACAR metamodel in order to handle the issue of ODDE autonomic systems validation. In Section 4, we expose the research work related to our proposition. Finally, we discuss our future work and conclude in Section 5.

2 Key Research Challenges

Deployment of complex software systems has become a nightmare for administrators. This deployment procedure essentially consists in accomplishing tasks to set up all middleware servers, as well as to deploy every business component upon these servers. In a fluctuating environment such as ODDE, it is impossible to manually perform these tasks, since the target machines hosting the applications are unknown. Moreover after the initial deployment, new machines can appear or disappear and manual administration intervention is needed. From this statement emerged the Autonomic Computing paradigm, which consists in extending programs with self-adaptation mechanisms. The core principles of the autonomic computing rely on the *control loop*, represented on Figure 1.

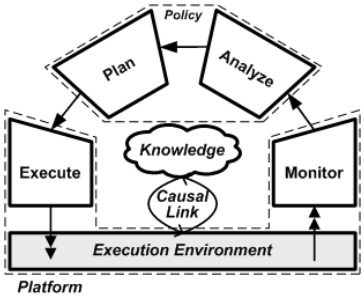


Fig. 1. The control loop of autonomic computing

This loop consists in four phases: Monitoring the system, analyzing the situation and taking a decision about some monitored changes, planning the adequate reconfiguration actions, and executing them. The analyzing and planning phases are relying on the *Knowledge* part as support for computation. It is an abstract representation of the autonomous system. At runtime the Knowledge part must always be conform to the execution environment, which means that every change in the execution environment must lead to an update of the Knowledge part, and *vice-versa*. Then a causal link must be maintained between the Knowledge part and the execution environment. Nevertheless, with the hypothesis of designing and deploying such autonomic systems are raised some issues, that are exposed in the following subsections.

2.1 Expression of Autonomic Behaviour

The first challenge to face considers the expression of autonomic behaviour of software systems. Adequate concepts must be identified for the administrators to express precisely and naturally their *autonomic policies*, *i.e.* the behaviour that they want to inject into their software architectures. Secondly, it is important that the paradigm for expressing software system behaviour at runtime is independent from any technologies. The mechanism that executes this behaviour must also be generic in order to apply this approach to software designed using any of these technologies. Finally, these concepts must be independent from the granularity of the software entity. Indeed, considering the deployment of whole software systems, administrators have to handle both with fine granularity business components, as well as with coarse granularity middleware servers. The autonomic behaviour paradigm must allow the administrator to write its autonomic policies for both of them.

2.2 Unit Deployment Validation

The first step to build an autonomic deployment process in an ODDE consists in writing procedures to install, configure and start pieces of software, and also procedures to stop, unconfigure and uninstall them. Autonomic mechanisms will then call these procedures at runtime according to the changes of the environment with respect to the global policy. Therefore, the first required validation concerns these procedures. Validating these procedures means ensuring that every instruction in a procedure (*e.g.* install, configure, start) must be cancelled in the opposite procedure (*e.g* uninstall, unconfigure, stop). This first step in validation then allows the administrators to write their autonomic policies using validated and safe deployment procedures for the different software involved in the system.

Here is a concrete simple example of such a problem. Considering a CORBA component-based application that has to be extended to every mobile phone entering the domain. On each mobile phone, a CORBA component server must be deployed, and then started (let's assume that this start procedure consists in launching a daemon). When this mobile phone leaves, a local autonomic policy must undeploy the component server and then launch the uninstallation procedure, which consists in removing the directory in which this component server was downloaded from the local filesystem. This action, of course, does not kill the component server daemon. Let's now consider that this mobile phone joins the domain again, and repeats this sequence (leave the domain, join the domain) several times: The CORBA component server will be started again several times on the mobile phone, and this can leads to a memory overflow, in that mobile phone.

2.3 Validation of Autonomic Policies

The last challenge concerns the autonomic policies themselves. These policies, according to the control loop, rely on the following principle: Then a *stimulus* occurs, under some *contextual conditions*, apply the adequate *reconfiguration actions*. However policies written using this paradigm can interfere with each other, as shown in [2], and there are many kinds of feature interactions. For instance, two different policies can be triggered by one unique stimulus, this is called the *Shared Trigger Interaction*. Another important feature interaction is the *Looping Interaction*: The reconfiguration action of a policy P1 can lead to the trigger of another policy P2 whose reconfiguration action leads to another succession of policy triggers that finally triggers the P1 policy. We fall into a cycle in policy execution. The list of feature interactions given here is not exhaustive. Using a strictly programmatic way to implement autonomic policies, it is impossible to detect such interaction between rules. So the challenge is to provide concepts that enable validation of policies.

Here is a concrete example of such a problem (represented on Figure 2). Three component types are involved in this example : `ClientComp` which has two required ports (logC and servC), `ServerComp` which has one provided port (servS) and `LogComp` which has one provided port (logS). We suppose that we have four autonomic policies expressed in an unformal paradigm:

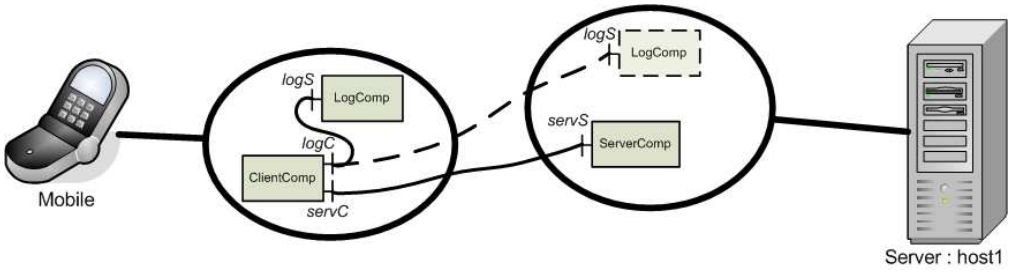


Fig. 2. An example involving cycling autonomic policies

RULE1 When a PDA enters the network, the deployment of a CORBA component server is launched and the deployment of a ClientComp component is performed on top of it.

RULE2 When a `ClientComp` (CC) is deployed onto a PDA, a remote binding is established between `CC.servC` and `SERVER.servS` (`SERVER` is a statically known instance deployed on the Host *host1*.)

RULE3 When a remote binding is made from a ClientComp instance (CC) required port, then a `LogComp` instance (LC) is deployed onto the PDA's component server (in order to log communications made through this binding), and a binding between `CC.logC` and `LC.logS` is established.

RULE4 When a remote binding is made from a `ClientComp` instance (CC) required port BUT there is no memory available to deploy the LogComp instance, then the `LogComp` instance (LC) is deployed on the Host *host1* and a binding between `CC.logC` and `LC.logS` is established.

One by one, these rules seem to be coherent and relevant to the application. Nevertheless a cycle can occur in the application of these rules: If the RULE4 is triggered, the binding between LC and CC becomes a remote binding, then the RULE4 is called again : the global autonomic behaviour falls into a cycle (for the sake of simplicity, the cycle here is very simple: a policy infinitely calls itself).

3 Our DACAR proposition

In this section we introduce some general details about our DACAR approach to execute autonomic system deployment in ODDE. DACAR allows to deploy complete software systems, from low-level installation of middleware servers to deployment of fine-grained business components.

3.1 Principles of the DACAR Approach

DACAR is based on a control loop, where the Knowledge model is implemented using models that abstract any relevant information about the underlying system. As can be seen on Figure 3, each part of the causal link between Knowledge and execution environment is implemented with two kinds of rules: The *Monitoring* and the *Deployment Rules*. The first one consists in monitoring the execution environment, and in case of change emerging from there, to reconfigure the Knowledge model. The second one consists in observing changes emerging from the Knowledge model and to apply these changes on the execution environment. The autonomic behaviour of the system is implemented through *Architectural Rules*. These rules represent the autonomic policies in our approach. As we have investigated in [4], models represent an adequate support to express all relevant information about the system, so fits well in the role of the knowledge model. We also identified that Event-Condition-Action (ECA) is a well tailored paradigm to express the causality between knowledge and execution environment as well as to express the autonomic behaviour.

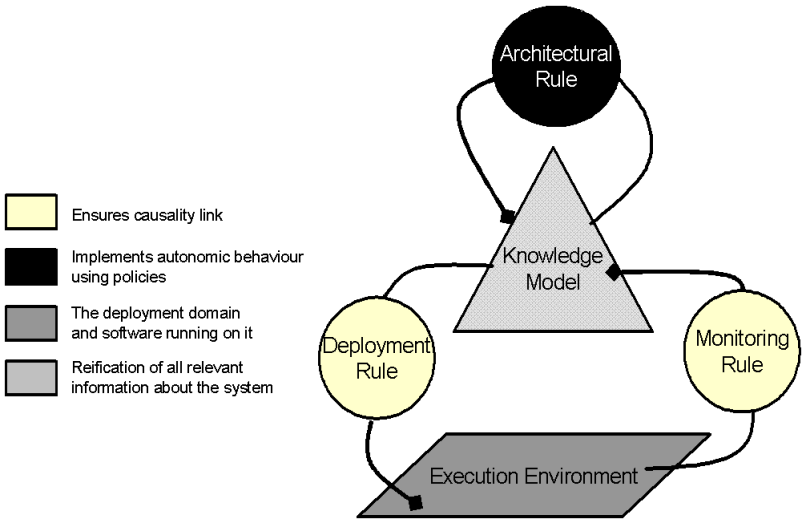


Fig. 3. Overview of the DACAR approach

3.2 Structure of the DACAR Metamodel

The first DACAR prototype presented a proof of feasibility of the approach, showing how to add autonomic behaviour in component-based software architectures (as it was CCM components). To achieve that, we have proposed the OMG D&C specification [5] as the Knowledge metamodel, but we found out that this specification only considers business components and does not allow multi-granularity deployment modeling. Moreover this specification encompasses several complex concepts that are not all useful for describing autonomic system deployment. In DACAR, it is possible to seamlessly express deployment and configuration of middleware servers as well as fine-grained business components, thereby a generic metamodel which exactly fits our needs has been established. Therefore our motivation is then to have a metamodel that focuses precisely on the deployment and autonomic concerns of a system, independently from the granularity chosen. This metamodel, represented on Figure 4 contains three subpart : the first one to define the deployment procedures of software systems, the second one to define the autonomic policies woven onto these systems, and the third only defines validation-specific concepts.

This metamodel expresses the main concepts about the deployment of several *Software* that are connected together to form a *System*, on low-left part of Figure 4. The deployment of a *Software* can depend on the deployment of other Software (*e.g.* a Java EE server depends on the Java runtime). Consequently, the sequence of deployment procedure will be scheduled according to these dependencies among Software. A Software is defined by several *Properties* such as the archive to download, where to install it, and other specific properties of the software. *Procedures* are also contained in a Software in order to install, maybe configure and start the Software as well as stop, uninstall the software or any other procedure specific to the Software deployed. These Procedures are composed of primitive *Instructions* to set environment variables, execute processes, etc. The *Host* is also represented in this metamodel in order to specify important access information about a specific target machine, such as file transfer (*e.g.* FTP) or remote access protocol (*e.g.* SSH). This Software metaclass is based on our another work which is called DEPLOYWARE. This work focuses on the execution of the deployment of a whole Software system according to a high-level description, and with respect to the dependencies of the system. This remains an ongoing work which is introduced in [6].

The first extension to this initial metamodel represents the answer to the first challenge announced in Section 2 and is about the expressivity of autonomic policies. Here we expose the autonomic part of our metamodel that allows the administrators to seamlessly express the autonomic policies of their systems

using ECA-like policies. This subpart of our metamodel is represented on the low-right part of Figure 4. A global autonomic behaviour of a *Software* element consists in a set of *Autonomic Policies*. A Policy is defined by an *Event* which is the concept that reifies a change occuring during the execution of the system. Under some *Conditions*, that depend on any element of the model, a set of *Actions* is triggered. This concept of action reifies any modification of the current model, which encompasses creation of a new software entity, reconfiguration of a property of a Software, or calling some Procedure of a Software. This metamodel is independent from any technology but also from any software granularity.

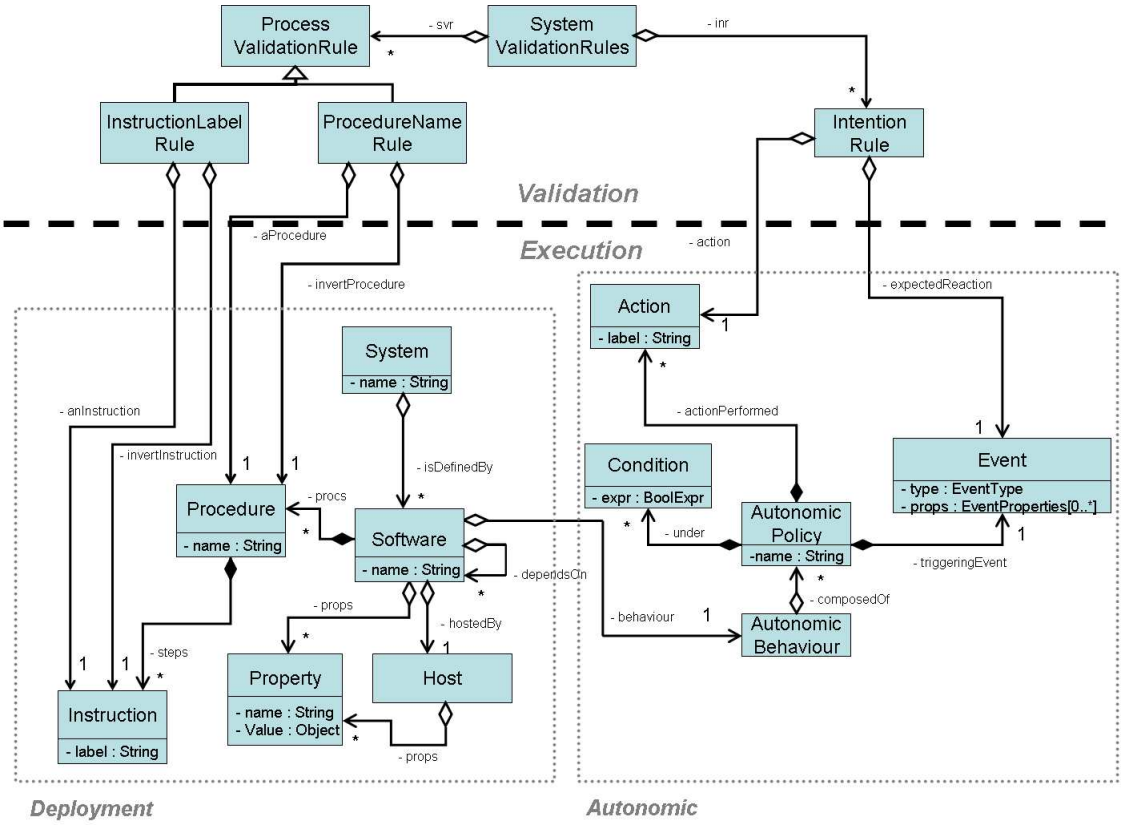


Fig. 4. The DACAR metamodel

3.3 Introducing Concepts in the Metamodel for Validation

In this section we expose the other concepts of the DACAR metamodel to answer the remaining challenges raised in Section 2. These validation-specific concepts are represented in the upper part of Figure 4.

Unit Deployment Validation — To face this challenge, we introduce the concept of *Process Validation Rule* (PVR). This particular type of rules is divided in two categories: The *Procedure Name Rules* (PNR) and the *Instruction Label Rules* (ILR). The first category allows the administrator to associate Procedures that have opposite goals. An example of PNR is the association between a `start` procedure and a `stop` procedure. Introducing such a rule ensures that, in a Software encompassing this Rule if the `start` procedure

is present, its invert stop procedure must also exist in the software. The ILR allows a more fine-grained verification into the procedures to inspect the instructions. This is an association between two opposite instructions. For example the instruction startProcess(P) (where P is the label of the process like `java myPackage.MyClass arg1` for instance) must be associated with the killProcess(P). Hence, using ILR it is possible to express that, for example, in a start procedure containing a startProcess(P) instruction, there must be a killProcess(P) instruction in the stop procedure. Consequently the combination of PNR and ILR ensures that every deployment actions can be cancelled completely without undesirable side effects.

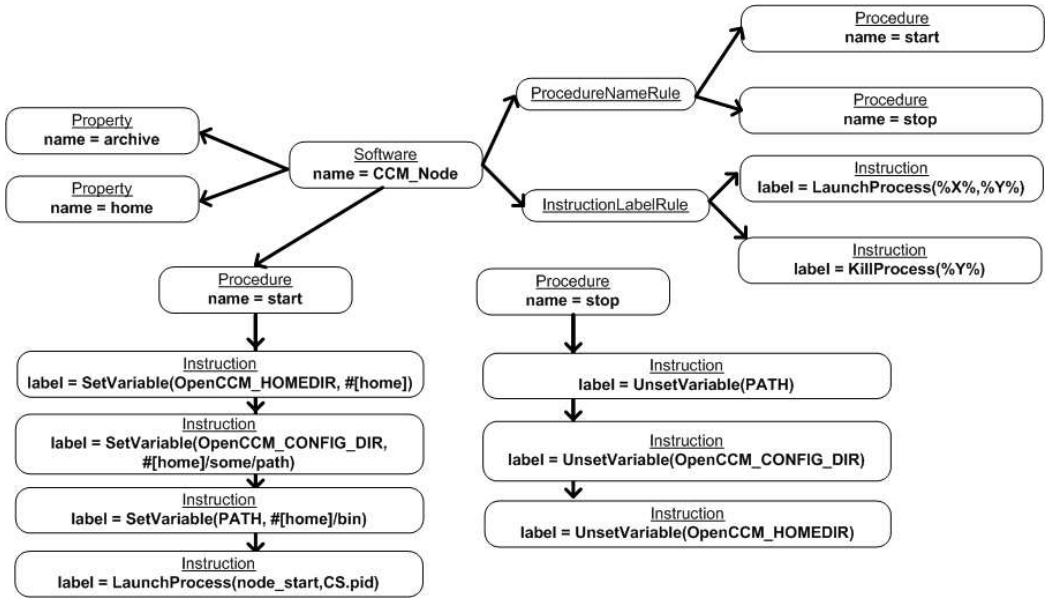


Fig. 5. Instance of a software description : A CORBA Component Server

In Figure 5 is represented an instance of Software representing the CCM component server (discussed in Section 2). Using introspection of this software, it is possible to check, for each action of the install process, and regarding the validation rules, to see if the invert instruction (or procedure) is present. In this case we can see that the `ProcedureNameRule` defined for the CCM server is respected since the software has a procedure start and also a procedure stop. Nevertheless, the `InstructionLabelRule` is not respected. There is, in the start procedure, an instruction labeled `LaunchProcess(%X%,%Y%)` (where `%X%` is the `node_start` command and `%Y%` is `CS.pid`). There should be, in the stop procedure, an instruction labeled `KillProcess(CS.pid)`. Then this software definition is not valid.

Validation of Autonomic Policies — Autonomic behaviour is expressed using rules in DACAR. Then, the problem of interactions between these rules is primordial and must be handled. For that, we introduce a last concept which is called the *Intention Rule* (InR) to achieve static validation of autonomic behaviour. This concept allows to associate a specific Event to an Action. This way the administrator expresses its intention: What change is expected to be performed when executing an action. Using this association of intention, it is possible to compute the sequence of rule triggers according to changes occuring at runtime. Figure 6 describes how we can analyse this sequence, in order to detect cycles, which is one of the most important harmful interaction between policies. From the list of changes likely to occur in the system, it is possible to get the different actions that compose the execution part of the policy. Then, thanks to the Intention Rule, it is possible to compute which events will then be produced due to the execution of these

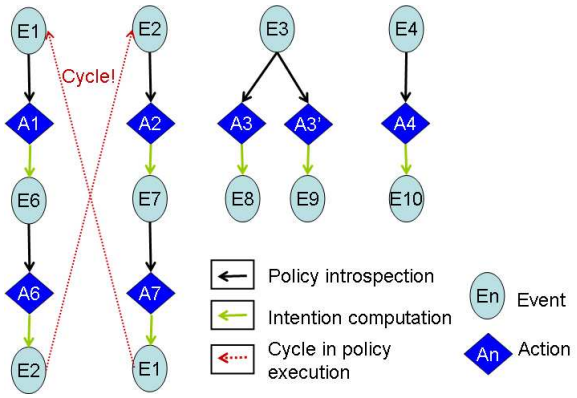


Fig. 6. An exemple of cycle detection thanks to the Intention concept

actions. Using this technique it is also possible to detect *Shared Trigger Interactions* which are detected using a graph as can be seen on Figure 6 when two vertices with different labels emerge from the same event.

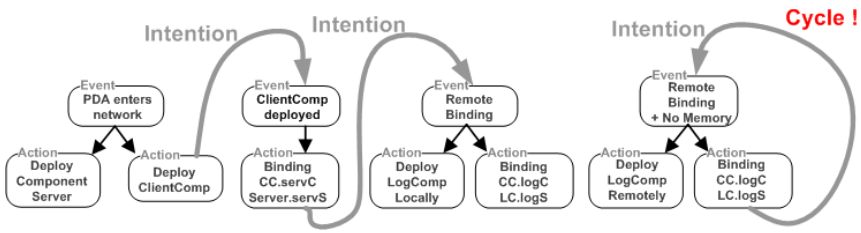


Fig. 7. Analysis of the rule intentions in an example

Figure 7 represents the autonomic policies of the component-based application enounced in Section 2: Using Intention Rules, it is possible to detect that the R4 rule is cycling.

4 Related Work

Jade proposes a component-based framework to build control loops to administrate J2EE applications on clusters [7]. The target platform and the application are modeled using Fractal components [8], in order to provide management interfaces. This allows the administrator to dynamically reconfigure the application architecture. Jade allows the architectures to be reconfigured according to infrastructure context changes. The Jasmine project [3], which strongly relies on Jade, offers an additional design layer to implement autonomic policies expressed using JBoss Rules [4]. This offers a convenient way to express rules, although no verification of the interaction between these rules are possible in contrast with DACAR.

The Rainbow framework [9] also implements a control loop to manage elements across the systems. It defines adaptation strategies using *invariants*, which are reconfiguration scripts executed in response to events. The use of invariants makes the policies in Rainbow monolithic, on the contrary of our approach. Consequently, this disadvantage leads to impossibility to detect interactions, and no verifications about the

[3] `http://jasmine.objectweb.org`
[4] `http://www.jboss.com/products/rules`

global behaviour of the system can be brought. The invariants are programs where only the syntax and the types employed can be statically verified. In addition of these two verifications, DACAR offers a behaviour validation which is crucial for long-life systems. Finally J2EEML is a modeling environment to implement autonomic EJB applications with QoS requirements [10]. These requirements are expressed using the graphical modeling environment and are then woven onto the components of the applications. Then, specific adaptation code is generated to make the EJB components able to be reconfigured according to the defined QoS requirements. This approach generates autonomic behaviour code from the defined model of QoS requirements, which leads to difficulties in introspection of the code, to validate the autonomic behaviour of the system as it is implemented in DACAR. Moreover, this approach seems to be specific to the EJB business components, and also specific to reconfigurations driven by QoS requirements: Reconfigurations due to fluctuations in an ODDE are impossible.

5 Conclusion and Future Work

In this paper, we have presented DACAR supporting model-based framework for autonomic heterogenous distributed software systems in ODDE. DACAR realizes the concepts of autonomic computing. By extending the DACAR metamodel with the adequate concepts we achieve a behaviour validation of autonomic policies. In this paper, two properties are ensured: The deployment and undeployment of Software are symetric, which means that no-side effects occurs when performing these two tasks. The second property is that autonomic policies are running safely, which means that no fatal interactions such as cycles are possible in the global autonomic behaviour. Indeed autonomic policies are classically expressed using programs or scripts, and no behavioural verification is possible, despite existing and well-identified problems in autonomic systems such as the feature interactions. DACAR allows the administrators to validate their whole software deployment thanks to two kinds of validation, the first one validates the correctness of the deployment procedures of the system, and the second one introspects autonomic policies to detect interactions between them. In this paper, two properties are ensured: The deployment and undeployment of Software are symetric, which means that no-side effects occurs when performing these two tasks. The second property is that autonomic policies are running safely, which means that no fatal interactions such as cycles are possible in the global autonomic behaviour.

A prototype of the DACAR metamodel has been developped using the KERMETA [11] metamodeling environment. Validation of several software as they are defined in our DeployWare framework has been successfully experimented. We are also driving experiments using Kermeta and DeployWare to provide an efficient, scalable and validated deployment framework for autonomic software systems. Our future work will mainly consist in consolidating autonomic deployment procedures validation in order to make autonomic deployment really effective and trusted. Another interesting work could be to find mechanisms to automatically infer Intention Rule from the specification of autonomic policies.

References

1. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Technical report, IBM Thomas J. Watson (2003) Published by the IEEE Computer Society.
2. Reiff-Marganiec, S., Turner, K.J.: Feature Interaction in Policies. Computer Networks 45 (2004) 569—584 Department of Computing Science and Mathematics, University of Stirling, United Kingdom.
3. Dubus, J., Merle, P.: Autonomous Deployment and Reconfiguration of Component-based Applications in Open Distributed Environments . In: Proceedings of the 8th International OTM Symposium on Distributed Objects and Applications (DOA'06). Volume 4277 of Lecture Notes in Computer Science, Montpellier, France, Springer-Verlag (2006) 26–27
4. Dubus, J., Merle, P.: Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In: Proceedings of the Models Workshop on Models@Runtime. Volume 4364 of Lecture Notes in Computer Science, Genova, Italia, Springer-Verlag (2006) 242—252

5. Object Management Group: Deployment and Configuration of Distributed Component-based Applications Specification. Available Specification, Version 4.0 formal/06-04-02 (2006)
6. Flissi, A., Merle, P.: A Generic Deployment Framework for Grid Computing and Distributed Applications . In: Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06). Volume 4279 of Lecture Notes in Computer Science, Montpellier, France, Springer-Verlag (2006) 1402–1411
7. Bouchenak, S., Palma, N.D., Hagimont, D., Taton, C.: Autonomic Management of Clustered Applications. In: IEEE International Conference on Cluster Computing, Barcelona, Spain, IEEE (2006)
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11-12) (2006) 1257–1284
9. Garlan, D., Cheng, S.W., Huang, A.C.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer **37** (2004) 46–54 2004.
10. White, J., Schmidt, D.C., Gokhale, A.: Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study. Volume 3713/2005. (2005) 601—615
11. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-Languages. In: Proceedings of MODELS/UML'2005. (2005) 264–278 Montego Bay, Jamaica.

Experiments with a Runtime Component Model

Jó Ueyama[1], Geoff Coulson[2], Edmundo R. M. Madeira[1], Thaís Batista[3], Paul Grace[2]

[1] Instituto de Computação (IC), Universidade Estadual de Campinas (UNICAMP)
13084-971 Campinas, SP – Brazil
[2] Computing Department, Lancaster University,
Lancaster LA1 4WA, UK
[3] Departamento de Informática (DIMAp),
Universidade Federal do Rio Grande do Norte (UFRN)
59072-970 Natal, RN – Brazil

Abstract. This paper provides a brief description of a general-purpose component model called OpenCom and also outlines the results obtained from the experiments with such a component model. The evaluation adopts a dual approach: first, we measure the inherent performance properties and overhead incurred by OpenCom. Then, we present three case studies which evaluate OpenCom's adaptive architecture in constructing system software for a wide range of domains.

1 Introduction

The component technology has been widely adopted to build general-purpose software at the application level by both the enterprise community and research community. For example, there are numerous component technologies for application development such as browser plugins, JavaBeans/Enterprise JavaBeans, the CORBA component model and Microsoft .NET. The success of the component approach comes from the benefits that are derived from componentisation, such as: *i)* it provides a higher degree of abstraction in software design, implementation, management and deployment; *ii)* it fosters third-party software reuse.

However, the notion of using components to build software at the *system level* (e.g. for building middleware platforms, or embedded systems) is less well established. In addition, the above mentioned benefits of componentisation appear as compelling as in this domain and this has been recognized by a growing amount of work using components for building system software. For example, Pebble [8] and Koala [12] propose the use of components for constructing embedded systems; OSKit [7], THINK [6] and MMLite [9] propose component-based OSs; proposals for component-based programmable networking environments include VERA [11], NetBind [2] and MicroACE [3]; proposals for middleware platforms include K-Component [5] and LegORB [13].

The main limitation with the above-mentioned component models is that they tend to be *narrowly targeted* and non-generic. This narrow targeting is clear in both of the following areas: *i)* the targeted systems for which they were designed (embedded systems, OSs, programmable networking environments or middleware platforms); for example, OSKit, Pebble and MMLite are exclusively targeted to build component-based operating systems; and/or *ii)* the environment at which they were intended to be deployed (e.g. most of the above mentioned models were deployed on conventional desktop machines as opposed to more non-conventional environments such as PDAs and embedded systems). As an example, VERA, NetBind and NP-Click are targeted to build programmable networking systems exclusively on the Intel IXP family routers [3].

This paper examines the experiments that were carried out with our general-purpose component model for building system software called OpenCom. Our component model supports adaptability which is demonstrated by three case studies that rely on OpenCom to build systems for a variety of domains and environments. The experiments with runtime reconfiguration demonstrates the feasibility of OpenCom in constructing reconfigurable systems.

In the remainder of the paper, Section 2 introduces the major features of the OpenCom component model. Following that, Section 3 provides the experiments with this technology. Finally, Section 4 offers conclusions and discusses how this work is taken further.

2 OpenCom's Key Features

It is important to highlight that Lancaster Open**COM** [4] refers to a previous component model targeted at middleware platforms, while Open**Com** concerns our new general-purpose component model. The former, i.e., Open**COM** is built on top of Microsoft COM and has been successfully applied to constructing re-configurable middleware platforms. However, the main limitation with OpenCOM is the inability to construct systems for a heterogeneous environment.

On the other hand, Open**Com** is aimed at constructing systems in a way that is independent of the deployment environment (e.g. heterogeneous environments like the IXP1200 routers (see Figure 2), which are resource constrained devices) and also aimed at constructing systems for a wide range of system domains (e.g. middlewares and operating systems). As a result, OpenCom consists of a more comprehensive programming model which is summarised below. Importantly, this is only a summary of the key features. A complete description of all features is found in [15].

The OpenCom's *kernel* is kept minimal and has only the support to deploy components of a particular style that is similar to XPCOM components. Component styles abstract between different system-level implementation of components – e.g. java components, C/++ components, assembly-language components. The kernel is adaptive and extra functionalities are all incrementally deployable as demanded at runtime.

In OpenCom, one builds systems by loading components and if required connects them to other components at runtime. The connection between components is called a *binding*.

The key motivation for *loaders* is to provide multiple-loading mechanisms in the underlying deploying environment. Loaders are OpenCom components that make a wide range of component styles be deployed in a heterogeneous environment, such as an embedded device. These loaders can encapsulate the complexity in loading components in such a heterogeneous environment.

Finally, *binders* are merely OpenCom components designed to provide a wide range of 'binding mechanisms'. Through binders, developers are free to implement any binding mechanism that might be required in the underlying deployment environment. As a concrete example of this, we implemented a binder that creates bindings between primitive Microcode components. This binder is employed in our first case study outlined in Section 3.

The notion of loader and binder gives support for adaptability as developers may utilize appropriate loaders and binders in order to construct systems for aimed environments.

3 Experiments with OpenCom

3.1 Overview

This section discusses performance evaluation and overhead incurred by OpenCom. It also talks about three case studies that use OpenCom to build systems for a veriety of domains. The aim of the case studies is to demonstrate the generality and adaptability of our component model.

The experiments outlined in this section were all carried out in a Dell Precision 340 series workstation with 4 x 1.6GHz Pentium CPUs, 512 MB of RAM, and running Linux Redhat 8.0. With regard to software, all the measurements were collected relying on an OpenCom kernel implemented in C++ for Linux.

3.2 Overhead and Performance

OpenCom kernel required a minimun memory overhead around 32Kbytes. In particular, this is comparable to the MMLite's kernel memory overhead which was measured as 26Kbytes. Such a memory overhead can enable us to deploy OpenCom in a wide range of resource constrained devices. As a consequence OpenCom is employed to construct sensor networking systems for running in the Motes and Gumstixs as outlined in [1].

The graph in Figure 1 compares the number of calls per second that was achieved by OpenCom bindings and direct C/C++ method calls. The OpenCom binding is an explicit receptacle that points to a virtual interface. This binding makes a clear separation between interfaces that lists required services and interfaces expressing provided services.

It indicates that OpenCom incurs a negligible overhead compared to that of C/C++ method calls. The number of calls per second achieved by OpenCom was obtained using two implementations of binding (i.e. one with v-table mediation and the other one without). Vtable (*virtual function table*) is a mechanism used in programming languages to enable dynamic polymorphism, i.e., run-time method binding. It is essentially a table containing pointers to virtual functions. This implementation is particularly common in languages such as C++ and C#. The compiler creates a separate vtable for each class and adds a pointer for each individual virtual function that is implemented in that class. Calls using v-tables are more expensive given that they need an extra reference to invoke the requested method.

3.3 Reconfiguration at Runtime

The imposed overhead to carry out reconfiguration at runtime is a critical issue, particularly for those with real-time requirements (e.g. audio and video multimedia systems with QoS requirements). The performance of three operations that are commonly employed for reconfiguring systems were measured to verify this overhead.

Operation	*Execution Time (μs)*
Component insertion	12.7
Component removal	3.3
Component replacement	16

Table 1. Average execution times for typical reconfiguration operations

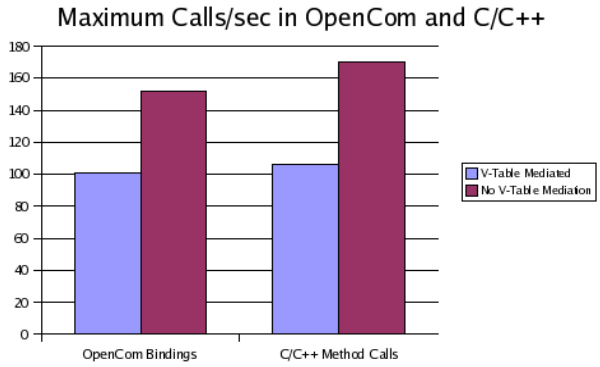


Fig. 1. Number of calls/sec in primary bindings and in C/C++ method calls

The insertion time consists of the time required to load, instantiate and create a binding between the components. This, in fact, refers to the worst case, which assumes that the requested component has not previously been loaded or instantiated. Thus, this figure can improve substantially if the requested component has previously been loaded.

The removal time refers to the time taken to unbind and destroy the requested component. This figure can also improve, if the requested component does not need to be destroyed. The time measured to destroy a binding between components was 3.1 μs. This figure includes the time taken to interact with the kernel registry, which includes the time for searching for the requested binding and then deleting it. The searching time has a linear increase/decrease as the kernel seeks the requested component in a linked list structure.

The above figure, which was measured to replace a component, includes the overhead incurred to deploy a new requested component and also the overhead imposed to unbind and destroy the 'old' component (i.e. the replaced component). Again, this figure can fall significantly if one only requires a replacement by unbinding the 'old' component and then creating a binding to an already-instantiated component.

The figures above demonstrate that OpenCom is suitable for systems that require a reconfiguration at runtime. For example, if one considers a typical QoS requirement of continuous media, in particular for audio and video streams, which require that delays should be no greater than 250 ms [14], the above figures seem feasible.

3.4 Case Studies

A qualitative analysis has been conducted to evaluate OpenCom in building systems independent of the target system and deployment environment. This has been verified by three case studies involving the construction of systems targeted at different domains.

The *first* case study applies the OpenCom programming model to the IXP1200 router environment [3] (see Figure 2). This router is particularly interesting for our research because it is *i*) heterogeneous (e.g. it has a number of processors, including the microengines that are specialised for packet processing); *ii*) resource poor having a small amount of memory; and finally *iii*) performance constrained (i.e. packets must be processed at line speeds). In terms of the implementation, we have developed a number of loader and binder components to deploy a wide range of component-styles in diverse environments. In particular, we

implemented a Microcode binder that uses the *code morphing* approach which was pioneered by the NetBind project [2]. Essentially, a component in Microcode is bound to another at runtime by rewriting a branch instruction so that execution jumps to the desired target.

In short, Figure 2 illustrates the outline of the IXP1200 based router, which is employed in the above case study. The router is composed of *i*) a StrongARM CPU running Linux, which acts as the general control processor in the router; *ii*) an array of six so-called microengines RISC CPUs that are attached to each other and share three types of memory shown in Figure 2 (i.e. Scratchpad, SDRAM and SRAM).

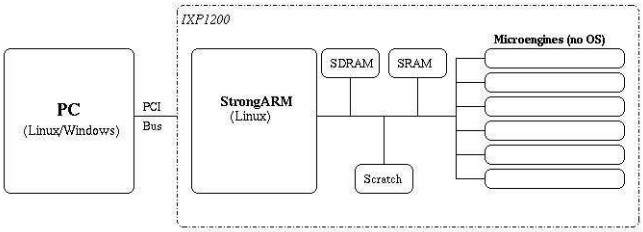


Fig. 2. Intel IXP1200 router architecture employed in our case study I

The *second* case study verifies the use of OpenCom to construct middleware for parallel environments called FlexPar. This research is funded by the São Paulo Research Council in Brazil. Essentially, this case study employs OpenCom to construct a flexible middleware that can be adapted according to the target parallel application. For example, if a developer desires to construct CSP (*Communicating Sequential Processes*) [10] based parallel software, appropriate loaders and binders are deployed to load and bind processes that implements the CSP Model. In short, CSP consists of processes and the communication mechanisms between them. It helps to avoid problems that are often encountered in multithreaded programming. In our experiments, the FlexPar kernel occupies only 60Kbytes which is suitable for most devices that runs parallel software.

Finally, in the *third* case study, OpenCom was employed to construct low-level software targeted at Sensor Motes. Sensor Motes are very primitive environments that consist of a small circuit board that hosts a number of electronic sensors and a simple 8-bit microcontroller. This case study aims to demonstrate that OpenCom can be applied in a way that is sufficient to construct low-level software running in resource-constrained devices such as a Sensor Mote. To deploy OpenCom in the Sensor Motes, the kernel had to be built on top of a simple microcontroller monitor program called *Contiki*. The implemented Contiki-kernel occupies only 30Kbytes, which demonstrates that it is portable to a wide spectrum of deployment environments. The sensor networking environment outlined in [1] also employs OpenCom to provide adaptability.

4 Conclusions and Further Work

This paper outlined the major experiments carried out with our general-purpose component model called OpenCom. The experiments demonstrated comparable results to that of other development tools such as C/C++. In addition, OpenCom incurred an overhead that is acceptable for systems that require runtime reconfigurations. The case studies in Section 3 demonstrated that the key features of OpenCom gave the

support for adaptability, which enabled OpenCom to build systems for a wide spectrum of domains and environments.

In our ongoing research we are looking at deploying OpenCom in a wider range of application domains and deployment environments. In particular we are investigating the use of OpenCom to build an architecture for networked embedded systems that encompasses dedicated radio layers, networks, middlewares, and specialised simulation and verification tools.

Acknowledgements

Jó Ueyama would like to thank the National Council for Scientific and Technological Development (CNPq - Brazil) for sponsoring his PhD scholarship at Lancaster University (Ref. 200214/01-2). The first and third author would also like to thank FAPESP for funding the FlexPar research project (Ref. 2006/06576-8).

References

1. *An Intelligent and Adaptable Flood Monitoring and Warning System*, September 2006.
2. A.T. Campbell, M.E. Kounavis, D.A. Villela, J.B. Vicente, H.G. de Meer, K. Miki, and K.S. Kalaichelvan. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *5th IEEE International Conference on Open Architectures and Network Programming (OPENARCH'02)*, June 2002.
3. Intel Corporation. Intel IXA SDK ACE Programming Framework Developer's Guide, June 2001. Part Number A71582-001.
4. G. Coulson, Blair G.S., M. Clarke, and N. Parlavantzas. The Design of a Highly Configurable and Reconfigurable Middleware Platform. *ACM Distributed Computing Journal*, 15(2):109–126, April 2002.
5. J. Dowling and V. Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In *Reflection 2001*, Kyoto, Japan, September 2001. LNCS 2192.
6. J.P. Fassino, J.B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX 2002 Annual Conference*, June 2002.
7. B. Ford, G. Back, G. Benson, J. Lepreau, A Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51. ACM Press, 1997.
8. E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. pages 267–282.
9. J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *8th ACM SIGOPS European Workshop*, pages 96–103, Sintra, Portugal, September 1998.
10. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
11. S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *4th International Conference on Open Architectures and Network Programming (OPENARCH)*, April 2001.
12. R. Ommering, F. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. 33(3):78–85, March 2000.
13. M. Roman, M. Mickunas, F. Kon, and R. Campbell. LegORB and Ubiquitous CORBA. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM2000)*, pages 1–2, Palisades, NY, USA, April 2000.
14. M. Salmony and H. Stuttgen. Transport services for multimedia applications on broadband networks. *Computer Communications*, 13(4):197–203, 1990.
15. J. Ueyama. *A Runtime Component Model for System Software*. PhD thesis, Lancaster University, 2006.

Endowing PauWare Components with Autonomic Capabilities

Cyril Ballagny, Nabil Hameurlain, Franck Barbier

LIUPPA, Avenue de l'Université, BP 1155, 64013 Pau, France
{ballagny,hameurlain,barbier}@univ-pau.fr

Abstract. In this paper, we discuss the extension of the PauWare component model [1] with autonomic capabilities. PauWare is a Java library for designing the inside of software components based on an execution engine of UML 2 State Machine Diagrams. Moreover, PauWare components communicate by means of event exchanges. Here, we propose to enhance this model with an internal feedback loop and, at the interaction level, with an interaction protocol. We first show how to specify the feedback loop elements and their relationships via a UML metamodel. Next, we show how to exploit an existing interaction protocol like the FIPA request one for improving the state machine event structure as well as the validation of the coordination of different feedback loops.

1 Introduction

The recent convergence between multi-agent systems and software engineering has led to agent-oriented software engineering [2]. Concomitantly, the emergence of autonomic computing [3] has created new research challenges about what agent systems may bring to software quality, the main concern of software engineering. For instance, self-adapting, self-configuring or self-healing software components are new kinds of components that have gained autonomy [4].

While the idea of autonomic computing encompasses the idea of self-managing or self-*[1] *individual* entities, the key problem is to define what could be self-managing communities. Even if this concept is widely explored in the field of autonomy-oriented computing and multi-agent system design research [5], it has not yielded clear and recognized notions in the area of software engineering. We mean what could be, for instance, a self-configuring assembly of components. At runtime, how to acquire new component services? How to replace component implementations by others without changing interfaces? Could interfaces (in terms of service signatures) dynamically change? May components interact differently (remote invocations, message passing) while they are supposed, contrary to software agents, to have more frozen collaborations. Once these problems are solved, what requirements these mechanisms may fulfill in terms of software engineering concerns?

1 In this paper, we assign to the idea of "self-*" or "self-management" all of the notions generally attached to autonomic systems, namely self-adaptation, self-configuration, self-protection, self-optimization, etc.

In the most widespread and well-known management technologies like JMX (Java Management eXtensions) and in the current research trends on autonomic computing, a great focus has been put on manageable or self-manageable infrastructures (*e.g.*, large distributed systems with wireless and wired parts) to the detriment of the characterization what should be self-managing fine-grained software elements. This paper puts forward the idea that models that persist at runtime (the concept of executable models) are appropriate instruments, to first design and next observe/control such entities. Having this characterization, self-* properties and policies may be derived from the model execution semantics. This paper aims at going beyond this existing autonomic framework by laying down and tackling the problem of self-* assemblies. For example, an optimization process launched in response to a load balancing demand, will consist in cloning some components, creating new interactions between these new components, moving some components from one deployment node to another, etc. As for self-optimization, this will suppose that components interact not only from a functional viewpoint but also from an application administration viewpoint in order to attenuate, in the spirit of autonomic computing, any human intervention.

After briefly describing how and why executable models depending upon UML 2 State Machine Diagrams, are appropriate supports for designing and running self-* components, this paper investigates the idea of self-* communities by gathering self-* individuals [6].

2 Autonomic Facilities in PauWare

A PauWare component is designed to behave at runtime by scrupulously respecting the semantics of UML 2 State Machine Diagrams (adoption of the run-to-completion mode especially). It is packaged into a component with distinguished interfaces and implementations. For instance, events labeling transitions become services of a provided interface. A component can request a service provided by another component sending it a broadcast event, *i.e.*, regardless of which state it is in and keeping its own work flow active.

2.1 Example of Autonomic Facilities: Lightweight Self-Configuration

In addition to the provided interface of the designed component, a configuration interface is proposed which can bear a "reset" service. In PauWare, this service may be easily and straightforwardly coded as follows:

```
public void reset() throws Statechart_exception {
        to_state("Idle"); // prerequisite: the component implements Manageable
}
```

The PauWare execution engine obviously forces the running component instance to be in the *Idle* state in a consistent way. The run-to-completion key principle of UML is respected: any non finished event processing is completed, any possible orthogonal state to *Idle* is carefully handled, etc. This action of dynamical re-configuration is subject to possible failures (the raising of a *Statechart_exception* instance in the code

above), leading to further re-configuration strategies (further details are described in [1]). In fact, the key issue is: Considering a self-* logic, which entity is responsible to request the "reset" service? The autonomic nature we expect supposes that either the running component which calls for re-configuration, itself calls the "reset" service or, any collaborating third-party component involved in a self-* policy calls it.

2.2 Self-Configuration as a Consequence of Self-Healing

PauWare supports rudimentary autonomic facilities for individuals. For instance, self-healing policies dedicated to a single component may be elaborated based on roll-back/undo actions which take advantages of states, transitions and state invariants. Rollback/undo actions correspond to reaching prior stable consistent states of the state machine (a set of orthogonal states) and re-computing invariants of these states to guarantee that the effect of actions associated with the cancelled transitions are themselves cancelled. In fact, self-healing processes first involve, of course, fault capture, analysis and diagnosis and next, imply various dynamical self-configurations to re-start, if feasible, the damaged software system.

3 Towards an Autonomic PauWare Component Model

The main idea of the proposed architecture is to endow PauWare components with autonomic capabilities. More precisely, a feedback loop is incorporated into each PauWare component. Next, Multi-Agent Systems interaction protocols such as those proposed by FIPA [8] are used for adaptive and dynamic coordination of self-* components together with their associated feedback loop.

3.1 Feedback Loop for Designing Self-* PauWare Components

The concept of feedback loop is fundamental in the design of self-managing systems [7]. In Figure 1, we show a UML metamodel which puts forward some extra mechanisms needed by a component to acquire autonomic capabilities. Thus, in PauWare, each autonomic component has its own control loop based on the following set of elements and their dependencies:

- A set of sensors that can detect both external and internal events such as anomalies like failures and so on;
- An aggregator which aggregates and formats information from all of its attached sensors. Then it transmits the information to its evaluator as well as to aggregators of distinct autonomic components (see the *provide-aggregate* self-association in Figure 1);
- An evaluator which encapsulates self-* policies and knowledge in order to choose the correct effector after analyzing the aggregator messages;
- A set of effectors which know how to act on their associated components for applying corrective actions such as *reset* and *rollback* actions.

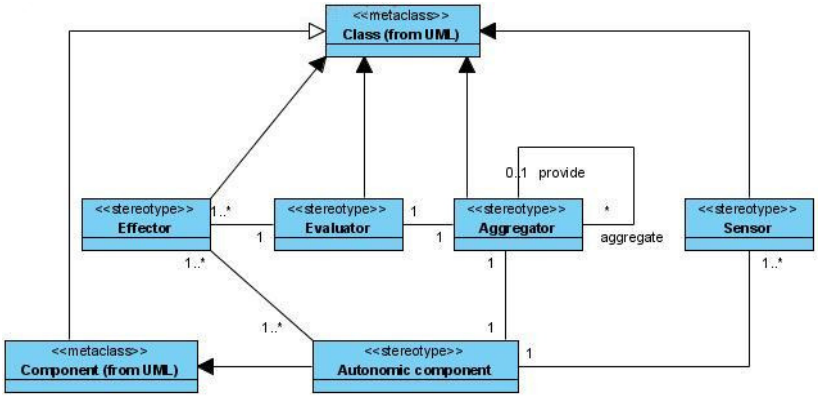


Figure 1. Autonomic PauWare component architecture.

Regarding the stability of the whole system, it is recognized that the composition of independent self-* entities can lead to an instable system even if the whole feedback loop consists of stable loops. Thus, starting from the description of the set of several well-formed individual autonomic components and their properties (specified by UML 2 State Machine Diagrams), the problem is to be able to ensure the stability of the whole system, and to deduce its global and emergent properties, *i.e.*, the properties of any autonomic component assembly. To tackle this problem, we propose to use MAS interaction protocols which make one agent to request another to perform some action, or more sophisticated protocols like the Contract-Net Protocol [8][2], for dynamic coordination of self-* entities together with their feedback loop elements. We briefly show in the next section how the FIPA-Request Protocol can be used for coordinating PauWare components together with their feedback loop and via effectors.

3.2 Interaction Protocols for Coordinating PauWare Components

In Multi-Agent Systems, agents use more sophisticated means of communication through structured messages and interaction protocols. A FIPA-ACL (Agent Communication Language) message structure enables the tracking of interactions inasmuch as the sender, the receiver, a replying queue, the protocol used, are included in the message. For instance, the *reply-to* parameter of the message is interesting in the sense that it enhances the communication to include other participants. Moreover, if we look to FIPA protocols, such as the Request protocol, they consider autonomy and fallibility of agents through, respectively, the *refuse* and *failure* communicative acts. For instance, in the Request protocol, the request can be refused by its recipient if the latter does not hold the necessary resources for succeeding or, if it is not in a state which allows the execution of this request. Then the sender of this request is able to envisage alternative procedures when it receives callback deny notifications. In this

[2] In order to ensure the stability of the whole system, we have assumed that self-* entities are organized in hierarchical way, so that is one feedback loop directly controls another feedback loop.

scope, events in UML 2 State Machine Diagrams should comply in PauWare with the FIPA message format.

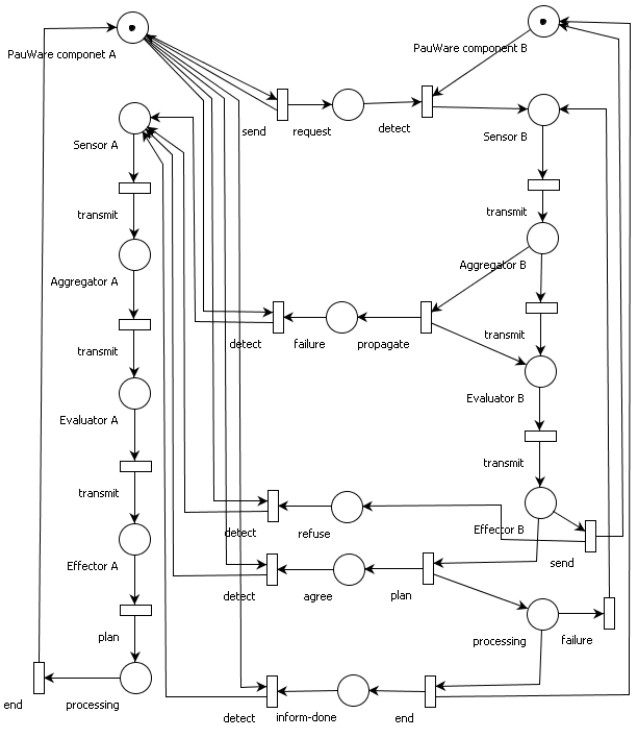


Figure 2. Proposed orchestration protocol for instrumenting self-* components.

In Figure 2, we give an example of Petri net-based specification which enables the interaction of two PauWare autonomic components conforming to the FIPA Request Interaction Protocol when one component requests a service from another. The request is internally processed by the control loop elements. In practice, a sensor detects this event, produces a record having the event details and submits it to the aggregator. Then, the aggregator gathers records and synthesize them in a report dedicated to the evaluator. This latter entity analyzes it and, according to self-* policies, selects the effector responsible for realizing the policy. Finally, the effector performs (executes) a *plan* (a set of actions) in order to recover a consistent state in the system and informs the autonomic component requesting the service about what has happened (request deny, internal failure, etc…). In this global context, the system stability is closely related to the termination of the in progress feedback loops.

4 Conclusion

The interaction between deployed PauWare components is until now, only based on standard communication means as supported by UML 2 Sequence Diagrams. Component collaboration occurs through synchronous/asynchronous communication (message passing, remote call…) using robust technologies like, for instance, Java Message Service (JMS) for having asynchronous capabilities. Even if the PauWare component model offers a rich composition support, it offers no possibility for formalizing and implementing self-* policies at the architecture level.

Due to this weakness, we sketch in this paper the required design patterns and canonical architectures for making components more autonomic. The existence and standardization of protocols for building autonomous communities led us to adopt FIPA protocols. However, in contrast with pure Multi-Agent Systems, we look for protocols that foster software quality at runtime: dynamical fault capture and recovering or self-healing, dynamical (re)-configuration and renewed deployment which match to self-configuration, etc. For that, our current framework and expertise about runtime models is a base from which self-* policies may be more formally described.

References

1. *PauWare* software and *PauWare* Users' Guide, available from: www.PauWare.com.
2. N. Jennings, "Agent-Oriented Software Engineering", Invited Paper, Proc. of MAAMAW'99, LNCS #1647, Springer, 1999, pp. 1-7.
3. J. Kephart, and D. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, 36(1), 2003, pp. 41-50.
4. M. Griss and G. Pour, "Accelerating Development with Agent Components," *IEEE Computer*, 34(5), 2001, pp. 37-43.
5. G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart, and R. White, "A Multi-Agent Systems Approach to Autonomic Computing," Proc. of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, Volume 1, *IEEE Computer* Society Press, 2004, pp. 464-471.
6. P. McKinley, S. Masoud Sadjadi, E. Kasten, and B. Cheng, "Composing Adaptive Software," *IEEE Computer*, 37(7), 2004, pp. 56-64.
7. P. Van Roy, "Self-management and the future of software design," Proc. Formal Aspects of Component Software (FACS '06), to appear in *ENTCS*, 2007.
8. FIPA, Geneva, Switerland, FIPA Specifications,http://www.fipa.org/specifications/index.html, 2002.

Modelling Adaptation Policies for Self-Adaptive Component Architectures

Franck Chauvel [1,2] and Olivier Barais [2]
[1] VALORIA, Université de Bretagne Sud
Campus de Tohannic, BP 573, 56017 Vannes Cedex, France
[2] IRISA, Université de Rennes 1
Campus de Beaulieu, F-35042 Rennes, France
Email: {prenom.nom}@irisa.fr

Abstract. In most of software systems, designers try to include some self-adaptation facilities to increase the reliability of their systems. However, despite of the new methods and technologies in software engineering such as CBSE, or AOP, it is still difficult to talk about adaptation since adaptation policies might impact the architecture, the configuration data, and some extra-functional features as well. We suggest in this paper a rule-based approach to model adaptation policies that enables the description of both architectural and functional adaptation and to relate them with extra-functional properties.

1 Introduction

Self-adaptations facilities are more and more used to fulfil some extra-functional requirements. In embedded systems, designers often use self-adaptation in order to maximize the reliability by adapting the system with respect to the available resources. However self-adaptation procedures are mostly designed and hard-coded in the same time when the initial design choices do not match with the extra-functional requirements. Two main reasons can explain this failure in the development process. On one hand, adaptation policies involved the description of both high level and low level extra-functional properties (such as reliability and memory consumption). For instance, it might be interesting to adjust the system in respect with the amount of available memory in order to adjust the reliability. These extra-functional properties are not easily handled on the design level and it makes difficult the expression of adaptation policies that are based on them. On the other hand, adaptation policies can impact both the architecture and the data configuration of the system. For example, in order to maintain the reliability of a web server, more data servers can be deployed, or the cache management policy can be changed to a more efficient one.

The contribution of our work is to enable a precise description of adaptation policies where both the architecture of the system and the configuration of particular components are impacted to involve extra-functional properties. An adaptation policy is thus described at two levels: at the architectural level using some imperative actions and at the functional level where the modifications of the configuration of a component are described using a rule-based approach.

The remainder of this paper starts with the introduction of a web server example to motivate the modelling of adaptation policies. Then, Section 3 shows how software architectures are modelled in order to enables the description of adaptation policies. Section 4 focuses on this particular point Finally, after having presented the main related works in Section 5, we sum up our contribution before discussing some future works in the conclusion.

2 Motivating Example

Let's consider a simple web server architecture that processes HTTP requests such as the Apache Web Server or the Microsoft IIS solution. One of the typical need of people who design such architectures, is to make it scalable. From the architectural point of view, it means that the architecture needs to self-adapt with respect to the load of the web server.

Because of the reliability requirements, we suggest a first solution of architecture that included a optional *cache* component and a set of data servers as shown on Figure 1. The incoming requests are handled by the *Proxy* component which can use the *Cache* component to solve the request or transfer it to the *Load Balancer* component. The request is then transferred to a data server and the answer is sent back to the *Proxy* component which deliver the related HTML page to the user.

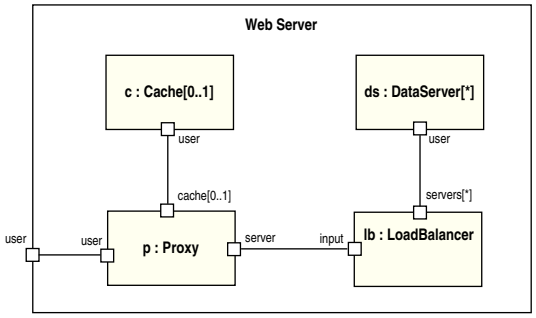


Fig. 1. The web server architecture modelled as UML2.0 component diagram

The designer adds the following requirements about the adaptation of its web server's architecture:

1. The cache must be used only if the number of similar requests is very high
2. The amount of memory devoted to the cache component must be automatically adjusted to the load of web server.
3. The validity duration of the data put in the cache must be adjusted with respect to the load of the web server.
4. More data server have to be deployed if the average load of the data servers is high.
5. The algorithm used to perform the load balancing must be changed according to load of the web server.

Here, requirements 1 and 4 are related to some architectural adaptations since it is required to update the architecture by adding (or removing) components. The others requirements are based on reconfiguration if we consider for instance that the data validity duration is a part of the configuration of the *cache* component. The approach presented in the following section allows designers to model adaptation policies which correspond to these requirements.

3 Modelling Component Architectures

This section presents briefly the component model used to described component-based architectures. In this component model, a *component* is an entity which interacts with its environment (other components) through well defined connection points named *ports*. A component is also an instance of a *component class* which defines the various features included in the component.

3.1 Modelling Primitive Components

A primitive component is a "basic" component: one which does not contain any other component. A component might interact with its environment in two ways: it can provide or require some services. Provided and required services are grouped into *interfaces* which are used to describe the different ports of a component. Figure 2 (left part) models the component class *Proxy* introduced in the web server example. This component class defines three ports, namely the *user* port, the *cache* port and the data port. Each port requires or provides interfaces.

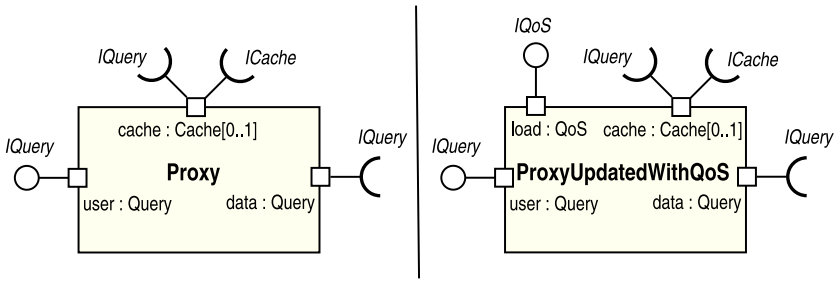


Fig. 2. The *Proxy* and *Proxy Updated* primitive components

3.2 Modelling Extra-Functional Properties

In order to include extra-functional properties in the architecture design our approach claims to reify in the architecture the sensor and the actuator (if they exist) related to these properties. For instance, to include the memory consumption of a component, we need to add a service which measures the memory consumption and another service which frees memory such as a garbage collector service.

In the example of the web server, since the designer needs to adapt the architecture with respect to the average load of the proxy, he has to define a *sensor* which is able to measure the average load. This *sensor* is thus reified as a service that can be added on any port of this component. Figure 2 (right part) shows the *Proxy* component class where a new port devoted to the quality of service purpose has been added. The service which measures the average load is defined in the interface named *IQoS*.

3.3 Modelling Component Collaboration

Primitive components can be put together to build more complex systems. A simple collaboration between two components is realized thanks to a connector that links one port of each of the two components. Connectors enables the description of complex collaborations which might be encapsulated into a *composite component*. For instance, the architecture shown by Figure 1 can be seen as a collaboration between a proxy component, a load balancer, a cache, and a collection of data servers.

In Self-adaptive architectures, the structure of the collaboration might change and the composite component has to handle this modification in order to keep the collaboration in a consistent state. To enable this, the interaction between the composite component and every sub components involved in the collaboration has to be described as all the others possible interactions: that is to say by using a specific port. In Figure 2 (right part) the port *load* will be used by a composite component which contains a proxy component to configure it.

4 Modelling Adaptation Policies

As shown in the motivating example, both architectural adaptation and reconfiguration might be needed. This section describes how to model this two types of adaptation in the component model previously described.

4.1 Architectural Adaptation

The first requirement of the motivation example sets that the cache component should be used only if the number of similar requests rises above a specific threshold. In our approach we suggest to add a sensor which measures the number of similar requests and to define an adaptation policy which adds (or removes) the cache component from the architecture with respect to the sensor's measurement. Since composite components are the entities which know all the participants of a collaboration, we suggest to make them responsible for the adaptation policy.

In order to describe such architectural adaptations we suggest to use an imperative language based on a set of architectural actions. These architectural actions are:

- **Create/Remove component instance** As we defined both component types and instances, we need to instantiate component types to get new instance and to remove old ones.
- **Create/Remove port instance** we defined also port types (with multiplicity) and so users need to create new port instances to support a new customer connection for instance.
- **Create/Remove slot instance** we also defined slot into composite components. Composite component types do not directly contain sub-component type, but a reference to another component type (called a *slot*) on which one can specify some multiplicity. So, we need a notation to create (or remove) slot instances into composite components.

– **Fill slot instance** During the life of a composite component, the contained-slot might be updated with a more efficient component. So we need to be able to fill slots.
– **Connect/disconnect port from connector** The most important aspects in component architecture is the composability of component that is reified by the notion of *connector*. So we need to connect (or disconnect) port instances form connectors.

Figure 3 shows the two operations required to describe the addition and the removal of the cache component. The *addCache* operation creates a new instance of the cache component class, put it into the devoted slot of the collaboration, and connects the two ad-hoc ports. The *RemoveCache* operation just breaks the connector that links the cache to the proxy component.

```
operation addCache() is
do
   self.cache := Cache.new
   connect(self.proxy.cache, self.cache.user)
end

operation removeCache() is
do
   disconnect(self.proxy, self.cache.user)
   self.cache.setEmpty
end
```

Fig. 3. An architectural adaptation described using architectural primitives

4.2 Modelling Functional Data Reconfiguration

The requirements 2 and 3 of the motivating example are related to component data configuration. In these two examples, it is required to update the configuration of the cache component with respect to the load of the web server. Since this kind of reconfiguration is mostly expressed as rules, we suggest to use a rule-based notation to stay as closed as possible of the requirements set by the designers.

The rules that are used to describe data reconfiguration are based on facts. Each rule links the state of a sensor to the state of an actuator. The left part of Figure 4 shows the two rules required to model the data reconfiguration of the cache component with respect to the load of the proxy component. The first rule set up that when the average load of the proxy component is “high” then the size of the cache is “big” and the validity duration is “long”. The other rule set up that when the average load of the proxy component is lower then the size of the cache is smaller and the validity duration is shorter.

To enable the use of terms such as “high”, “low”, “small” or ”big”, we consider that the sensor and the actuator used to reify every extra-functional properties (such as the size of cache, or the validity duration) offer services that define these terms. For instance, the cache component can offer a port that enables the configuration of the size of its memory with two services such as setLowMemory and setHighMemory. The mapping between these services and the rules can be ensured thanks to naming rules for such services.

```
proxy.averageLoad is 'High' =>
   cache.size is big
   and cache.validityTime is '
       Long'

proxy.averageLoad is 'Low' =>
   cache.size is 'Small'
   cache.validityTime is 'Short
       '
```

```
mode WithCache is
   trigger is proxy.SimilarRequestNumber is
       'High'
   entry is addCache()
   exit is removeCache()

   proxy.averageLoad is 'High' =>
      cache.size is 'Large'
      and cache.validityTime is 'Long'

   proxy.averageLoad is Low =>
      cache.size is 'Small'
      cache.validityTime is 'Short'
end
```

Fig. 4. An reconfiguration-based adaptation described using rules

4.3 Mixing Architectural Adaptations and Reconfigurations

In order to mix these two kinds of requirements in a single notation, the whole adaptation policy has to be defined in terms of modes. Each mode reflects one state of the architecture and includes all the rules which manage the functional data configuration.

Each mode refers to the architectural reconfigurations needed to switch (and to switch back) to this particular architectural mode. It defines also the rules that manage the functional data configuration of the components involved in this architectural mode. The right side of Figure 4 shows the mode that is defined to control the addition and the removal of a cache component into the web server architecture.

In this example, to enter the *WithCache* mode, the web server component (the composite component that manage the whole collaboration) has to run the *addCache* operation. The shift from the *normal* mode to the *WithCache* mode is triggered by a rule which set that the shift is performed when the number of similar requests detected on the proxy is “High”. This information comes to the composite component thanks to the service which reifies the sensor. This service must be available on the port which connect the *web server* composite component to *proxy* component.

5 Related Work

Many tools have been developed in recent years to manage architectural adaptation or re-configuration at run time in component-based software [1, 2] . However, there is still a gap between the modelling level (where designers specify component, data, and behaviour) and the implementation level where component are running.

Various Architecture Description Languages (ADLs) have been developed in the past as shown in [3]. However, most of them only describe software architecture in a static way. Recently several works have shown the interest of describing the software architecture dynamic [4]. For example, AADL [5] allows designers to model different modes of the same system, where each mode represents a particular state of the architecture evolution but there is no way to describe how the system switches between two modes.

In [6], Allen and al. suggest a first way to manage architectural re-configuration in component-based model. They extend the Wright component model to design component assemblies and perform consistency and verifications. This extension reuses the behaviour notation of Wright to model the reconfiguration. It allows the architect to view the architecture in terms of a set of possible architectural snapshots, each with its own steady-state behaviour. Transitions between these snapshots are accounted by special reconfiguration-triggering events. To introduce the dynamism in an architecture description, the architect has to modify the component's alphabet, and allow new messages to occur in port descriptions. Through this approach, the interface of a component is extended to describe when reconfigurations are permitted in each protocol in which it participates. Thanks to these new events, a "reconfiguration view" consumes these events to trigger reconfigurations. Contrary to our approach, they mainly work to represent some events that trigger the reconfiguration. In our work, we consider that any event can trigger the reconfiguration and we specify the reconfiguration policy at the composite level with a set of predefined modes.

Some component models have also been developed to describe component architectures. As seen in [7] most of them only deal with the structure of the component architecture. For instance, SOFA [8, 9] use CSP to describe behaviour protocols on ports and enables static verification. But the behavioural description only deals with signal emissions and signal receptions and no syntax is related to the architecture reconfiguration.

Rainbow [10] provides another appraoch to talk with component-based self-adaptation. This approach is very closed to our but has not been designed to support architectural reconfiguration and data reconfiguration in the same time.

In Fractal [11] user can defined some specific *controllers* on composite components to manage the internal of the component. Thus, adaptation policies can be described thanks to these controllers. However, controllers are a very low-level mechanism and they are related to the implementation. Our approach allows designers to express adaptation policies in the early stages of the design process.

6 Conclusion

In this paper, we introduce a way to model adaptation policies into component-based architecture. The contribution of our approach is to take into account both the architectural adaptation and the functional data reconfiguration. To do this, the description of adaptation policies is based on the definition of several modes for the architecture. Each mode corresponds to a particular state of the architecture and can be related to some extra-functional properties measured in the architecture. The designer specifies a trigger condition for each mode which specifies when the system have to switch from one mode to this one. He specifies also the rules that manage the functional data adaptation inside each mode. This work is a first step in order to manage adaptation policies in early design steps in component-based architectures. However, it is still difficult to design the modes. From the same architecture state, a different order of the adaptation modes can lead to different architectures. We plan to express the whole behaviour of each component in order to simulate the architecture and to predict performance for self-adaptive architectures such as memory consumption.

References

1. Batista, T.V., de La Rocque Rodriguez, N.: Dynamic reconfiguration of component-based applications. In: PDSE. (2000) 32–39
2. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In Morrison, R., Oquendo, F., eds.: Software Architecture, 2nd European Workshop, EWSA 2005, Pisa, Italy, June 13-14, 2005, Proceedings. Volume 3527 of LNCS., Springer (2005) 1–17
3. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. In: IEEE Transactions on Software Engineering. Volume 26. (January 2000) 23
4. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, New York, NY, USA, ACM Press (2004) 28–33
5. SAE, A..E.C.S.C.: Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506 (November 2004)
6. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. Lecture Notes in Computer Science **1382** (1998)
7. Lau, K.K., Wang, Z.: A survey of software component models (April 2005) Preprint CSPP-30, School of Computer Science, The University of Manchester.
8. Plásil, F., Bálek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: CDS '98: Proceedings of the International Conference on Configurable Distributed Systems, Washington, DC, USA (1998)
9. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11) (2002) 1056–1076
10. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
11. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: An open component model and its support in java. In Crnkovica, I., Stafford, J.A., Schmidt, H.W., Wallnau, K., eds.: Component-Based Software Engineering: 7th International Symposium, CBSE 2004. Volume 3054 of LNCS., Springer Verlag (Jan 2004)

A Reconfiguration Mechanism for Statechart Based Components

Xabier Elkorobarrutia[1], Goiuri Sagardui[1], and Xabier Aretxandieta[1]
xelkorobarrutia,gsagardui,xaretxandieta@eps.mondragon.edu

Mondragon Unibertsitatea, Computing Department, Arrasate, 20500, Spain

Abstract. This paper describes a reconfiguration mechanism for software components based on hierarchical state-machines. This mechanism can be applied, among other uses, to component self-healing, component composition and adaptation. We introduce a framework in order to development statechart based components that allows modification of the component's model at run-time. To accomplish this, we transform the model employed at design-time in a reflexive architecture of the component. This framework is well suited for model driven style of development; furthermore it makes the component *"aware"* of its structure without the involvement of the developer. This work focuses on application-independent mechanisms for software components run-time reconfiguration.

1 Introduction

Autonomic Computing (AC) in general, and Self-Healing in particular, have gained much attention but it's not very well defined in terms of scope ([7] [6]). In this area, most of many works have been oriented to highly distributed and heterogeneous corporate systems, where the main obstacle to further progress in the Information Technology industry is the inability to manage the system as a whole. In [10] a roadmap to accomplish this problem is proposed and the core of the proposal is to make the software self-aware and give it the ability to self-manage.

The characteristics of autonomic systems have been further classified into the so called *"self-*"* properties; and in order to accomplish them, it is necessary to monitor the system. Many works have shown the ability of some techniques to contribute to the materialization of AC. Besides the need of system monitoring, most of them present one of these two commonalities: reconfiguration is achieved by means of the overall component architecture change, and, the actuation in each individual component is accomplished through predefined actuators. Our work aims at extending this latter characteristic, providing an application independent way of actuating in a component.

Basically, the reconfiguration of a system of components is acomplished by the replacement, replication, reubication, ... of components; and middlewares provide mechanism to facilitate it. But in order to reconfigure each individual component of a system, fewer general mechanisms have been proposed. Usually they are application-dependent and must be taken into account by the designer. As an example in which no developer involvement is needed, [4] has defined a way to introduce self-healing functionalities in java legacy code in a non intrusive way. [5] has described a framework to dynamically attach a repair engine to a managed application, without hardwiring it into the application and without crosscutting it. These mechanisms employ language constructs (classes, operations, types ...) as the units with which they operate. This is too low level for our aim because we want to operate in a component at the model level.

[8] proposes the usage of runtime models for a systematic self-management. Those runtime models are designed and generated according to specific needs. In our work, we present a framework for developing statechart based software components whose objective is twofold: assist the developer in the transformation of a statechart model into an implementation and, to create such transformation in a way that makes the model available and modifiable at run time. This is achieved by creating a reflective architecture of the component that permit us to reconfigure the component in the same language that we have used at

modeling; in our case, statecharts. When designing the component, little effort is required to take into account reconfiguration aspects, because the used framework implicitly will add those reconfiguration abilities.

We will use statecharts for the construction of active objects that have a message receptor and a message dispatcher. Those messages are sent asynchronously. In particular, it can be used for implementing some kinds of software components. We are experimenting with JAVA applications with one statechart inside and employing CORBA for message exchanges.

2 Statechart Implementation

To implement state machines, lot of patterns have been proposed, each of them with their pros and cons. Multiple criteria can be applied: memory consumption, execution overhead, easiness of development, extensibility .... Most of these patterns put emphasis on design time. For example, one of the most popular implementations, Quantum[9], is an optimal C++ implementation of statecharts, but does not offer any support at run-time. In [3] the *Reflective State* pattern is described, but its aim goes toward extensibility, reusability and fault-tolerance support.

Although the runtime support is the goal for our framework, we also want from it to support a Model Driven style of development. This work is based on Pauware framework[1] due to its possibilities to be modifiable at runtime. Our framework imposes some transformation rules that implicitly creates an object structure that reflects the statechart. Thus, changing the statechart could be achieved by changing this object structure.

Let us illustrate it with the simple example of Fig. 1. An extract of the corresponding generated code is shown in listings 1.1 and 1.2. The former reflects the state structure and the latter its behavior: what action and when should be executed, but not how. And the same applies for guards.

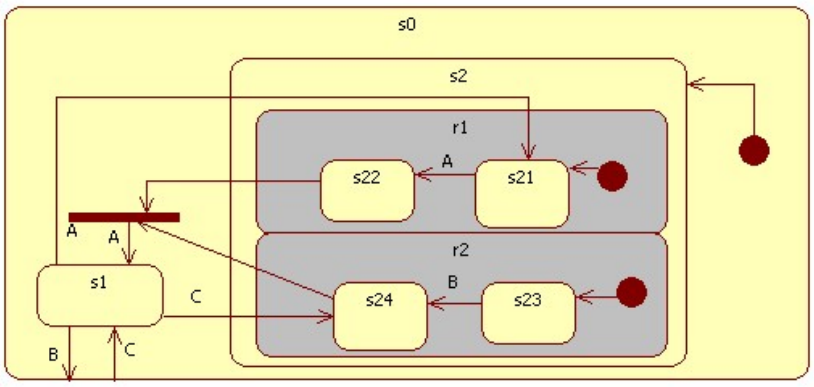


Fig. 1: A statechart example.

```
root=new XorState();
rootState=root;
s1=new XorState();
root.addState(s1);
s2=new AndState();
root.addInitialState(s2);
reg1=new Region();
s2.addRegion(reg1);
s21=new XorState();
.........
```

Listing 1.1: Structure definition of Fig. 1.

```
jr=new JoinReaction(s1,null,"evA_join");
jr.addSource(s22);
jr.addSource(s24);
s2.addReaction(EvA.class,jr);
s1.setExitAction("exit_1");
.......
```

Listing 1.2: Behavior definition of Fig. 1.

The actions/guards will be executed/evaluated by a different part of the component, the *"executor"*, as shown by Fig. 2 (Fig. 3 shows some classes for the implementation of some parts). Whenever a message arrives to the dispatcher, if the state machine is stationary, it gets the actual active states and interrogates them for what should be done. In order to obtain that, those states ask the executor to evaluate some guards. After the determination of the actions that have to be carried on, the dispatcher gives instructions to some states and transitions to execute their exit, entry or whatever action they must execute; and finally, those are delegated to the executor part. The *"context"* part is a global repository of the statechart where the actual active state configuration and the processing message is stored. The reason to this last decision is that if we want to modify the model at runtime, the statechart definition part should be independent of the message parameters and its actual values. This avoids hardwiring some logic in it and therefore, diminishing the options to change.

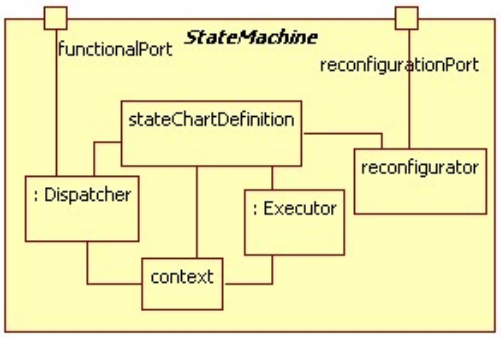


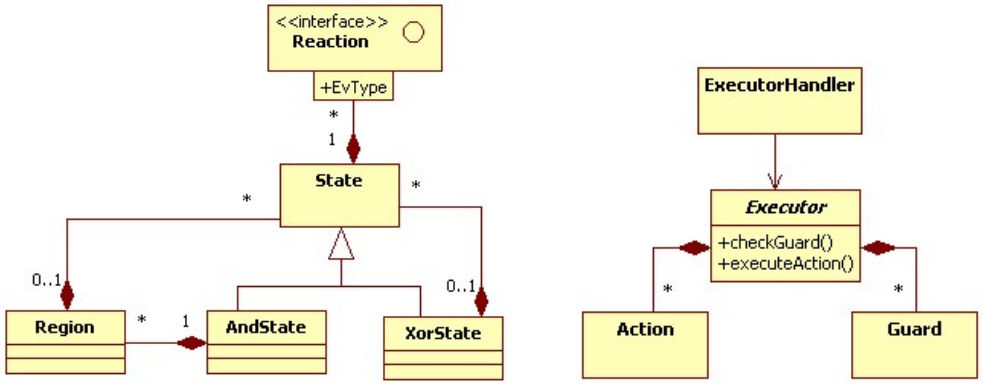
Fig. 2: The state machine structure.

3 Reconfiguration Possibilities

The executor part of the state machine has no state except some pseudostate variables[9] as counters and references. The total separation of a statechar definition and the actions/guards associated to it make possible to change them separately. Next, we describe some uses of those changes.

3.1 Software Fault-Tolerance

N-versioning and Recovery-Block approaches are general mechanisms to cope with software faults that need to be particularized for each application. In [2] there is an extensive field study and classification of software



(a) Classes for statechart definition part.

(b) Executor part structure.

Fig. 3

faults. In this study, the assignment, checking and algorithm defects are around 82%. These kinds of defects are the ones that could be made in the guard and action definition of statecharts. In the presented statechart structure, all this is gathered in the action executor part. Fig. 3b illustrates its structure. There is an extra indirection through the `executorHandler` element and it is the place where we can put interceptors, catch exceptions and change the executor in the presence of software-faults. In the event of a fault detection, among other choices, the executor handler would restore the previous pseudostate variables and retry with the new version of the executor. It is a particularization of the Recovery-Block aproach. Of course, we are assuming that when talking about software faults, each statechart dependent code is much less tested than the framework in which it is based.

About the mechanism used to implement this executor, there are many choices: it could be a single class or we could use another granularity. For example, one class per state. Those and other choices are possible but in our case, a simple class has been used. The different versions of the executor will only have to cope with implementing the actions and guards of the statechart, not taking into account the statechart model.

3.2 Adaptation of Components

When constructing a system with COTS(Commercial off-the-shelf) software components, there is a need for fine tuning their collaboration. And in this scenario, it is impossible that the developer of the component could anticipate all the different scenarios where it is going to run. Through parametrization, we could achieve some of that tuning but this involves an additional complexity that the developer must cope with.

Our framework supports the changing of the statechart model at run-time: add new states, eliminate transitions, etc. To make that possible, the statemachine must have an additional port. We will ask the component for those changes through this reconfiguration port by means of a meta-object protocol. This mechanism has the benefit that the component developer does not need to contemplate any situation where the control part is needed to change. This is delegated to the system integrator that, anyway, must do it. Limitations arise if we need to add an additional action or guard not contemplated before. In this case, another complementary mechanism is required.

Another kind of adaptation is a *"mode change"* (e.g. from automatic to manual mode). In order to lower the complexity of the software, and provided that the executor part has an enough action repertory, this mode change can be as simple as model changing. We could make a statechart to instruct the framework to change entirely the model, transmutating thus the behavioral part specified by the statechart of the software component.

3.3 Composition

When composing a system of components, there are some system-wide requisites that must be accomplished among many of the components. For example, if we have purchased a component that controls the windows of room and another that controls the heating, we have multiple choices to combine them to obtain a temperature control system. And it is not possible to anticipate them when designing each of the components. For example, *"the heating system will not start to run if a window is open"*. Or, *"The system must close the window if the heating system starts to run"*.

There are many choices in deciding how to implement those requisites. One solution would be the modification of the original components adapting them to each particular situation. If the framework for constructing the statecharts offers basic facilities of sending messages and interrogating about its actual active states, at initialization time we could add conditions like `not(window.inState("close"))` or put actions like `window.putMesagge("close")`. This enables us to modify a software component by adding a behavior not contemplated at design time.

4 Conclusions and Future Work

We have presented a framework for the development of statechart based components that, in addition to support a Model Driven style of development, offers among other characteristics, a built-in ability to change the statechart model at run-time. This reconfigurability is based upon an application independent *"language"*; in our case, statecharts. The component developer is exempted to cope with reconfigurability issues when working with functional aspects and therefore, software complexity is lowered.

A component reconfiguration plan can be defined in terms of the basic facilities offered by the framework.

At this moment, in the statechart definition, the framework allows to employ AND/OR states, regions, transitions, forks, joins, but does not allow to employ other UML statechart elements e.g. some pseudostates, or transition segments. Future work points in two directions: to extend the statechart elements that the framework supports; and to define a set of modification possibilities that would be enough for an extensive set of cases and determine its limitations.

References

1. Barbier Franck: MDE-based Design and Implementation of Autonomic Software Components. International Conference on Cognitive Informatics, ICCI (2006)
2. Duraes Joao A., Madeira Henrique S.: Emulation of Softwatre Faults: A Field Data Study and a Practical Approach: IEEE Transactions on Software Engineering vol **32** (2006)
3. Ferreira Luciane Lamour, Rubira Cecilia M.F.: The Reflective State Pattern. Pattern Languages of Programs PLoP'98 (1998)
4. Fuad M. Muztaba, Deb Debzani, Oudshoorn Michael J.: Adding Self-Healing Capabilities into Legacy Object Oriented Applications. International Conference on Autonomic and Autonomous Systems ICAS '06 (2006)
5. Griffith Rean, Gail Kaiser: Manipulating Managed Execution Runtimes to Support Self-healing Systems. Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005)
6. Kephart Jeffrey O.,Chess David M.: The Vision of Autonomic Computing. Computer **36**(2003)
7. Philip Koopman: Elements of the Self-Healing Problem Space. Workshop on Software Architectures for Dependable Systems(WADS2003)(2003)
8. Rohr Matthias, Boskovic Marko, Giesecke Simon, Hasselbring Wilhelm: Model-driven Development of Self-managing Software Systems. Models in Software Engineering (2006)
9. Samek Miro: Practical Statecharts in C/C++: An Introduction to Quantum Programming. CMP Books (2002)
10. White Steve R., Hanson James E., Whalley Ian, Chess David M., Kephart and Jeffrey O.: An Architectural Blueprint for Autonomic Computing. First International Conference on Autonomic Computing (ICAC'04) (2004)